Programming in VLSI:

From Communicating Processes To Delay-Insensitive Circuits

Alain J. Martin

Department of Computer Science
California Institute of Technology

Caltech-CS-TR-89-1

PROGRAMMING IN VLSI:

FROM COMMUNICATING PROCESSES TO DELAY-INSENSITIVE CIRCUITS

Alain J. Martin

to appear: *UT Year of Programming Institute on Concurrent Programming*
C.A.R. Hoare, editor; Addison-Wesley, 1989

The research described in this report was sponsored by the Defense Advanced Research Projects Agency, ARPA Order Numbers 3771 and 6202; and monitored by the Office of Naval Research, under contract number N00014-87-K-0745

Department of Computer Science
California Institute of Technology
Pasadena, CA 91125

Caltech-CS-TR-89-1

Programming in VLSI

From Communicating Processes to Delay-Insensitive Circuits

1

Alain J. Martin
California Institute of Technology

Delays have dangerous ends.
—William Shakespeare

Introduction

With chip size reaching one million transistors, the complexity of VLSI algorithms —i.e., algorithms implemented as digital VLSI circuits— is approaching that of software algorithms— i.e., algorithms implemented as code for a stored-program computer. Yet design methods for VLSI algorithms lag far behind the potential of the technology.

Since a digital circuit is the implementation of a concurrent algorithm, we propose a concurrent programming approach to digital VLSI design. The circuit to be designed is first implemented as a concurrent program that fulfills the logical specification of the circuit. The program is then compiled —manually or automatically— into a circuit by applying semantic-preserving program transformations. Hence, the circuit obtained is correct by construction.

The main obstacle to such a method is finding an interface that provides a good separation of the physical and algorithmic concerns. Among the phys-

ical parameters of the implementation, *timing* is the most difficult to isolate from the logical design, because the timing properties of a circuit are essential not only to its real-time behavior, but also to its logical correctness if the usual synchronous techniques are used to implement sequencing.

For this reason, *delay-insensitive* techniques are particularly attractive for VLSI synthesis. A circuit is delay-insensitive when its correct operation is independent of any assumption on delays in operators and wires except that the delays be finite [17]. Such circuits do not use a clock signal or knowledge about delays.

Let us clarify a matter of definitions right away: The class of entirely delay-insensitive circuits is very limited. Different asynchronous techniques distinguish themselves in the choice of the compromises about delay-insensitivity.

Speed-independent techniques assume that delays in gates are arbitrary, but that there are no delays in wires. *Self-timed* techniques assume that a circuit can be decomposed into *equipotential* regions inside which wire delays are negligible [16]. In our method, certain local "forks" are introduced to distribute a variable as inputs of several operators. We assume that the differences in delays between the branches of the fork are shorter than the delays in the operators to which the fork is an input. We call such forks *isochronic* [6].

Although we initially chose delay-insensitive techniques for reasons of methodology, those techniques present other important advantages in terms of efficiency and robustness:

- The clock rate of a synchronous design has to be slowed to account for the worst-case clock skews in the circuit and for the slowest step in a sequence of actions. Since delay-insensitive circuits do not use clocks, they are potentially faster than their synchronous equivalents.
- Since the logical correctness of the circuits is independent of the values of the physical parameters, delay-insensitive circuits are very robust to variations of these parameters caused by scaling or fabrication, or by some nondeterministic behavior such as the metastability of arbiters. For instance, all the chips we have designed have been found to be functional in a range of voltage values (for the constant voltage level encoding the high logical value) from above 10V to below 1V.
- Delay-insensitive circuit design can be modular: A part of a circuit can be replaced by a logically equivalent one and safely incorporated into the design without changes of interfaces.
- Because an operator of a delay-insensitive circuit is "fired" only when its firing contributes to the next step of the computation, the power

consumption of such a circuit can be much lower than that of its synchronous equivalent.

Since the correctness of the circuits is independent of propagation delays in wires and, thus, of the length of the wires, the layout of chips is facilitated.

The method indeed produces correct and efficient circuits. It has been applied, with both "hand" compilation and automatic compilation, to a series of difficult design problems such as distributed mutual exclusion, fair arbitration, routing automata, stacks, and serial multipliers. All fabricated chips have been found to be correct on "first silicon". Although our CMOS implementation of the basic operators has been overly cautious, and the electrical optimization techniques have been rather tame, the performance of the chips has been found at least equal to that of synchronous implementations. We have just completed the design of a general-purpose microprocessor, and its performances are very encouraging: In $1.6\mu m$ SCMOS, it runs at 18 million instructions per second. (See the conclusion, Section 23, for more detail.)

The main reason for the efficiency of the method is that, rather than going in one step from program to circuit, the designer applies a series of transformations to the original program. At each stage, powerful algebraic manipulations can be performed leading to important optimizations in terms of speed or area.

In the first part of this chapter, we present the "source code" notation, the "object code" notation, and a VLSI implementation of the production rules in CMOS technology. The source notation is inspired by C. A. R. Hoare's CSP [4]: A program is a set of concurrent processes communicating by input and output commands on channels. (A similar experience in the use of communicating processes for programming in VLSI is described in [13].) The object code notation, called *production rule set*, is one of the main innovations of the method and is an interesting notation for digital VLSI all by itself.

In the second part, we describe the four main steps of the compilation (process decomposition, handshaking expansion, production rule expansion, operator reduction), illustrating them with a number of examples. In particular, we present the different algebraic transformations that can be applied at different stages of the compilation and that give the method its flexibility and efficiency.

Part I: The Source Code and the Object Code

1 *The Program Notation*

For the sequential part of the notation, we use a subset of Edsger W. Dijkstra's guarded command language [3], with a slightly different syntax. We give only an informal definition of the constructs' semantics.

(i) $b\uparrow$ stands for $b :=$ **true**, $b\downarrow$ stands for $b :=$ **false**. Those assignments are called "simple assignments".

(ii) The execution of the *selection* command $[G_1 \rightarrow S_1 [] \ldots [] G_n \rightarrow S_n]$, where G_1 through G_n are boolean expressions, and S_1 through S_n are program parts (G_i is called a "guard", and $G_i \rightarrow S_i$ a "guarded command"), amounts to the execution of an arbitrary S_i for which G_i holds. If $\neg(G_1 \vee \ldots \vee G_n)$ holds, the execution of the command is suspended until $(G_1 \vee \ldots \vee G_n)$ holds.

(iii) The execution of the *repetition* command $*[G_1 \rightarrow S_1 [] \ldots [] G_n \rightarrow S_n]$, where G_1 through G_n are boolean expressions, and S_1 through S_n are program parts, amounts to repeatedly selecting an arbitrary S_i for which G_i holds and executing S_i. If $\neg(G_1 \vee \ldots \vee G_n)$ holds, the repetition terminates.

(iv) *Sequencing*: Besides the usual sequential composition operator '$x; y$', we introduce two other operators. For atomic actions x and y, 'x, y' stands for the execution of x and y in any order leading to termination. For noninterfering communication actions x and y, "$x \bullet y$" stands for the simultaneous execution of x and y. (We shall return to this definition when we discuss the implementation of communication in Section 19.)

(v) $[G]$, where G is a boolean expression, stands for $[G \rightarrow$ **skip**$]$ and thus for "wait until G holds". (Hence "$[G]; S$" and $[G \rightarrow S]$ are equivalent.)

(vi) $*[S]$ stands for $*[$**true** $\rightarrow S]$ and thus for "repeat S forever".

(vii) From (ii) and (iii), the operational description of the statement

$$*[[G_1 \rightarrow S_1 [] \ldots [] G_n \rightarrow S_n]]$$

is "repeat forever: wait until some G_i holds; execute an S_i for which G_i holds".

(viii) Tail recursion is allowed, but not general recursion. Functions and procedures with a simple parameter mechanism are also used, but we will not discuss them here.

1.1 *Communicating Processes*

A concurrent computation is described as a set of processes composed by the usual concurrent composition operator ||. The concurrent composition is *weakly fair*; i.e., if, in a given state of the computation, x is the next atomic action of one of the processes, then x will be executed after a possibly unbounded but finite number of atomic actions from other processes.

Processes communicate by communication actions on *ports*; they do not share variables.[1] A port of a process is paired with a port of another process to form a *channel*. When no messages are transmitted, communication on a port is reduced to synchronization signals. The name of the port is then sufficient to identify a communication action.

If two processes, $p1$ and $p2$, share a channel with port X in $p1$ and port Y in $p2$, at any time the number of completed X-actions in $p1$ equals the number of completed Y-actions in $p2$. In other words, the completion of the nth X-action "coincides" with the completion of the nth Y-action. If, for example, $p1$ reaches the nth X-action before $p2$ reaches the nth Y-action, the completion of X is suspended until $p2$ reaches Y. The X-action is then said to be *pending*. When, thereafter, $p2$ reaches Y, both X and Y are completed. The predicate "X is pending" is denoted as $\mathbf{q}X$. If, for an arbitrary command A, $\mathbf{c}A$ denotes the number of completed A-actions, the semantics of a pair (X, Y) of communication commands is expressed by the two axioms:

$$\mathbf{c}X = \mathbf{c}Y \tag{A1}$$

$$\neg\mathbf{q}X \vee \neg\mathbf{q}Y \tag{A2}$$

Surprisingly, it is possible (and even advantageous) to define communication actions as coincident and yet implement the actions in completely asynchronous ways.

1.2 *Probe*

Instead of the usual selection mechanism by which a set of pending communication actions can be selected for execution, we provide a general boolean command on ports, called the *probe*. The definition of the probe given in [5] states that in process $p1$, the probe command $\overline{X}$ has the same value as $\mathbf{q}Y$. For the time being, we use a weaker definition, namely:

$$\overline{X} \Rightarrow \mathbf{q}Y$$

$$\mathbf{q}Y \Rightarrow \Diamond\overline{X},$$

1. We have made a restricted use of shared variables in the design of the microprocessor.

where $\diamond P$ means *P holds eventually.* (We will return to the first definition in the example on the implementation of a fair arbiter.)

1.3 *Communication*

Matching communication actions are also used to implement a form of distributed assignment statement, to "pass messages", as it is often said. In that case, the pair of commands is specified to consist of an input command and an output command by adjoining them to the symbols "?" and "!", respectively. For example, $X?$ is an input command and X is therefore an input port, and $Y!$ is and output command and Y is therefore and output port.

Axiom Communication axiom
Let $X?u$ and $Y!v$ be matching, where u is a process variable and v is an expression of the same type as u. The communication implements the assignment $u := v$. In other words, if $v = V$ before the communication, then $u = V$ and $v = V$ after the communication.

1.4 *First Example: Port Selection*

Process *sel* repeatedly performs communication action X or communication action Y, whichever can be completed; *sel* is blocked if and only if neither X nor Y can be completed:

$$sel \equiv *[[\,\overline{X} \rightarrow X \,[]\, \overline{Y} \rightarrow Y\,]]\,.$$

Obviously, process *sel* is not fair because of the nondeterministic choice of a guard when both guards are true. Negated probes make it possible to transform *sel* into a fair version, *fsel*:

$$\begin{aligned} fsel \quad \equiv \quad & *[[\,\overline{X} \rightarrow X;\ [\overline{Y} \rightarrow Y \,[]\, \neg\overline{Y} \rightarrow \textbf{skip}] \\ & \quad []\ \overline{Y} \rightarrow Y;\ [\overline{X} \rightarrow X \,[]\, \neg\overline{X} \rightarrow \textbf{skip}] \\ &]]\,. \end{aligned}$$

Negated probes are necessary for implementing fairness.

1.5 *Second Example: Lazy Stack*

We implement a stack S of size n, $n > 0$, as a string of n communicating processes defined as follows:

$$S = \begin{cases} h, & \text{if } n = 1, \\ (h \| T), & \text{if } n > 1, \end{cases}$$

where h, the head of the stack, is a process, and T, the tail of the stack, is a stack of size $n-1$. Process h communicates with the environment of the stack by the communication actions $in?x$ and $out!x$, and with T by the communication actions $put!x$ and $get?x$. Hence, $h.put$ matches $T.in$, and $h.get$ matches $T.out$. (We assume that no attempt is ever made to add a portion to a full stack, or to remove a portion from an empty stack.)

Each stack element either is empty and behaves like program E, or is full and behaves like program F. The epithet "lazy" is attributed to this stack because no reshuffling of portions takes place after a portion has been removed from a full stack element.

$$\begin{aligned} E \equiv\ & [\overline{in} \rightarrow in?x;\ F \\ & [] \overline{out} \rightarrow get?x;\ out!x;\ E \\ &] \end{aligned}$$

$$\begin{aligned} F \equiv\ & [\overline{out} \rightarrow out!x;\ E \\ & [] \overline{in} \rightarrow put!x;\ in?x;\ F \\ &]. \end{aligned}$$

The following alternative coding of the stack element process, due to Peter Hofstee, illustrates the advantages of the probe construct:

$$\begin{aligned} & *[[\overline{in} \rightarrow in?x \\ & \quad [] \overline{out} \rightarrow get?x \\ & \quad]; \\ & \quad [\overline{out} \rightarrow out!x \\ & \quad [] \overline{in} \rightarrow put!x \\ & \quad]]. \end{aligned}$$

We assume that each stack element is initially empty.

2 *The Object Code: Production Rules*

Carrying the discrete model of computation down to the transistor level requires that the MOS transistor be idealized as an on/off switch. Unfortunately, the simple semantics of the switch ignore too many electrical phenomena

that play an important role in the functioning of the circuit. A crucial innovation of the method is that the transistor need not be viewed as a discrete switch; voltages can change continuously from one stable level to the other one, provided that the changes are monotonic.

The notation for the object code provides the weakest possible form of control structure and the smallest possible number of program constructs. In fact, it contains exactly one construct, the *production rule* (PR), and one control structure, the *production-rule set.*

We consider the production-rule notation to be the canonical representation of a digital circuit. This representation can be decomposed into several equivalent networks of digital operators, depending on the set of building blocks used, but the production-rule set represents the circuit independently of the chosen implementation.

Definition A PR is a construct of the form $G \mapsto S$, where S is either a simple assignment or an unordered list "$s1, s2, s3, \ldots$" of simple assignments, and G is a boolean expression called the guard of the PR.

Example

$$x \wedge y \mapsto z\uparrow$$
$$\neg x \mapsto u\uparrow, v\downarrow$$

The semantics of a PR are defined only if the PR is *stable*:

Definition A PR $G \mapsto S$ is said to be stable in a given computation, if, at any point of the computation, G either is **false** or remains invariantly **true** until the completion of S.

Stability is not guaranteed by the implementation. It has to be enforced by the compilation procedure.

Definition An execution of the stable PR $G \mapsto S$ is an unbounded sequence of *firings.* A firing of $G \mapsto S$ with G **true** amounts to the execution of S. A firing of $G \mapsto S$ with G **false** amounts to a **skip**.

Definition A PR set is the concurrent composition of all PRs of the set.

2.1 *Operations on PR Sets*

The only composition operation on two PR sets is the set union.

Theorem
The implementation of two concurrent processes is the set union of the two PR sets implementing the processes and of the PR sets implementing the channels between the processes, if any.

The proof follows from the associativity of the concurrent composition operator.

The other operations on the PRs of a set are those allowed by the following properties:

Multiple occurrences of the same PR are equivalent to one as a consequence of the idempotence of the concurrent composition.

The two rules $G \mapsto S1$ and $G \mapsto S2$ are equivalent to the single rule $G \mapsto S1, S2$.

The two rules $G1 \mapsto S$ and $G2 \mapsto S$ are equivalent to the single rule $G1 \vee G2 \mapsto S$.

2.2 *Noninterference*

We require that *complementary* PRs —i.e., PRs of the type $G1 \mapsto x\uparrow$ and $G2 \mapsto x\downarrow$— be *noninterfering*.

Definition Two complementary PRs are noninterfering when $\neg G1 \vee \neg G2$ holds invariantly.

It can be proven that, under the stability of each PR and noninterference among complementary PRs, the concurrent execution of the PRs of a set is equivalent to the following sequential execution:

$*$[*select a PR with a true guard; fire the PR*]

where the selection is weakly fair (each PR is selected infinitely often). From now on, we ignore the firings of a PR with a **false** guard; a firing will mean a firing of a PR with a **true** guard.

Until we return to these issues, we shall assume that the stability and noninterference requirements are fulfilled.

3 *VLSI Implementation of PRs*

Stability and noninterference are the two properties that make the VLSI implementation of PRs (almost) straightforward. As an example, we describe how PRs can be implemented in CMOS technology.

3.1 *The CMOS Transistors*

A CMOS circuit is a network of "nodes" —variables— interconnected by transistors. Certain nodes are also connected to the input-output "pads", which provide the interface with the environment; we will ignore the pads in this presentation. Other nodes are directly connected to the *power* node, providing the constant high-voltage value —called *VDD*— that represents the logical constant **true** or 1. Yet other nodes are directly connected to the *ground* node —called *GND*— providing the constant low-voltage value that represents the logical constant **false** or 0.

A node takes the continuous range of voltage values between the high voltage and the low voltage. Above a certain voltage $v1$ the value is interpreted as 1. Below another voltage $v0$, the value is interpreted as 0. Thanks to the stability property, the precise values of $v1$ and $v0$, which vary from node to node, are irrelevant provided that $v0 < v1$ and the voltage changes are *monotonic*. (Strict monotonicity is not necessary and is actually impossible to achieve because of noise, but we will not enter into these details here.)

A CMOS transistor is of either n-type or p-type. A transistor relates three nodes in the following way. Let g, standing for "gate", and x and y be the three nodes. When g is **false** for an n-transistor, and **true** for a p-transistor, no current passes through the region between x and y, called the *channel*;[2] thus x and y are left unchanged.

When g is set to **true** for an n-transistor, or **false** for a p-transistor, the channel becomes conducting. In this case, either x and y have the same voltages and are left unchanged, or a current is established in the channel until x and y reach the same voltage. The common value reached by x and y depends on electrical properties of x and y that are determined by the physical sizes (capacitances) of the nodes implementing x and y and by their interactions with the rest of the circuit. (Differences in node capacitances may cause charges to flow through the channel of a transistor in a way that results in unintended values of the nodes. This phenomenon, called *charge sharing*, may make it quite difficult to predict the final voltage value reached by x and y.)

In order to define the net effect of a PR independently of the physical parameters of its implementation, we are going to restrict the use of transistors. (In particular, the restriction will eliminate most occurrences of charge sharing.)

We impose the condition that a transistor used in isolation connect only two variables of the circuit: the gate g and one of the other two nodes, say z.

2. This notion of channel is unrelated to the one we introduced for communication among processes.

The third node of the transistor is either the power or the ground. With this restriction, the behavior of a single n-transistor is

$$g \mapsto z\uparrow \quad \text{or} \quad g \mapsto z\downarrow.$$

The behavior of a single p-transistor is

$$\neg g \mapsto z\uparrow \quad \text{or} \quad \neg g \mapsto z\downarrow.$$

3.2 *Threshold Voltages*

The current in the channel of a transistor is a function of the so-called gate-to-source voltage, V_{gs}, defined as $V(g) - min(V(x), V(y))$ for an n-transistor and as $V(g) - max(V(x), V(y))$ for a p-transistor. In first approximation, the current is assumed to be zero when

$$V_{gs} \leq V_{tn}$$

for an n-transistor and

$$V_{gs} \geq V_{tp}$$

for a p-transistor. V_{tn} and V_{tp} are called the *threshold voltages.* (Typically, $V_{tn} \approx 1V$ and $V_{tp} \approx -1V$.)

Because of the existence of threshold voltages, if an n-transistor is used to implement $g \mapsto z\uparrow$, the final value of z is not a "strong" 1, since the channel will stop conducting as soon as the voltage of z is within V_{tn} of the gate voltage. And symmetrically, a p-transistor used to implement $\neg g \mapsto z\downarrow$ does not produce a "strong" zero as the final value of z. Since the voltage drops caused by the threshold voltages accumulate as we compose operators, it is important to produce strong signals in order to be able to compose an arbitrary number of operators. We shall therefore restrict our use of n-transistors to PRs of the form

$$g \mapsto z\downarrow \tag{1}$$

and p-transistors to production rules of the form

$$\neg g \mapsto z\uparrow. \tag{2}$$

With these restrictions, all implementations produce strong signals.

Threshold voltages are difficult to adjust in CMOS technology. Actually, they tend to become more variable as the feature size decreases. (They may also vary during the activity of the circuit because of some electrical interaction with the substrate, called *body effect.*) For constant node capacitance,

variations in thresholds account for most of the discrepancies in propagation delays on a CMOS chip. In particular, these variations exclude the possibility that the ordering in space of a set of variables along a common wire be used to infer an ordering in time of a set of transitions of these variables.

3.3 *Switching Circuits*

Consider the canonical (stable) PR

$$b \mapsto z\downarrow, \tag{3}$$

where b is a boolean expression in terms of a set of variables. These variables are used as gates of transistors implementing a switching circuit s corresponding to b: s is a series-parallel switching circuit between the ground node and z. The switches are n-transistors whose gates are the variables of b, possibly negated. Furthermore, we have

$b \equiv$ *"there is a path from ground to* z *in* s*"*.

By the construction of s, if b holds and remains stable, z is eventually set to 0. (For this reason, s is called a *pull-down circuit*.) Hence, s is exactly the implementation of production rule (3).

Using a symmetrical argument, we can show that the same series-parallel circuit as s, but with the power node and z connected, and whose switches are p-transistors, implements the production rule

$$bneg \mapsto z\uparrow, \tag{4}$$

where *bneg* is derived from b by negating all variables. (This circuit is called a *pull-up circuit*.)

4 *Operators*

Two PRs that set and reset the same variable, such as

$$\begin{aligned} b1 &\mapsto z\uparrow \\ b2 &\mapsto z\downarrow, \end{aligned} \tag{5}$$

are implemented as one operator.

Let $s1$ be the pull-up circuit corresponding to $b1$, and let $s2$ be the pull-down circuit corresponding to $b2$. The two circuits are connected through the common node z (see Figure 1). Since noninterference has been enforced, $\neg b1 \vee \neg b2$ holds at any time. This guarantees the absence of a conducting path

between power and ground when the operator is not firing. (A path may exist for a short time when the operator is firing.)

Definition The operator implementing the two rules is called "combinational" if $b1 \vee b2$ holds at any time, and "state-holding" otherwise.

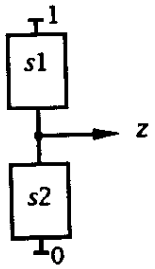
By definition, if (5) is combinational, there is always a conducting path between either *VDD* or *GND* and the output z. Hence, the value of the output is always a strong 0 or a strong 1, and therefore $s1$ and $s2$ are together a valid implementation of (5).

For example, PRs (1) and (2) together implement an inverter as represented in Figure 2. The circuit of Figure 3 implements the *nand*-operator defined by the PRs

$$a \wedge b \mapsto z\downarrow$$
$$\neg a \vee \neg b \mapsto z\uparrow .$$

If (5) is a state-holding operator, $\neg b1 \wedge \neg b2$ may hold in a certain state. In such a state, node z is isolated; there is no path between z and either *VDD* or *GND*. In MOS technology, an isolated node does not retain its value forever; eventually the charges leak away through the substrate and also through the transistors of the pull-up and pull-down circuits. If the PRs of the operator are fired frequently enough to prevent leakage, the implementation of Figure 1 can be used for a state-holding operator. Such an implementation is called *dynamic*.

Figure 1. CMOS implementation of a combinational operator.



Otherwise, it is necessary to add a storage element to the output node of a state-holding operator. Such an implementation is called *static*. In the sequel, we assume that only static implementations are used for state-holding operators.

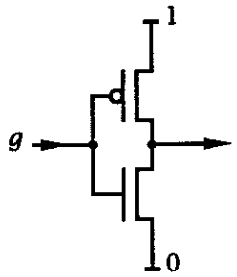
(A standard CMOS implementation of such a storage element consists of two cross-coupled inverters (see Figure 4). This implementation inverts the value of z. The "weak" inverter, marked with a letter w on the figure, connects z to either *VDD* or *GND* through a high resistance, so as to maintain z at its intended voltage value [18].)

The implementation of a static state-holding operator is slightly more costly than that of a combinational operator because of the need for a storage device. Hence, given a pair of PRs that are not combinational, we may first try to modify the guards —under the invariance of the semantics— so as to make them combinational.

5 *The Standard Operators*

All operators of one or two inputs are used, and are therefore viewed as the standard operators.

Figure 2. A CMOS inverter.



5.1 *One-Input Operators*

The two operators with one input and one output are the *wire*:

$$\begin{aligned} x \underline{w} y \quad \equiv \quad & x \mapsto y\uparrow \\ & \neg x \mapsto y\downarrow , \end{aligned}$$

and the *inverter*:

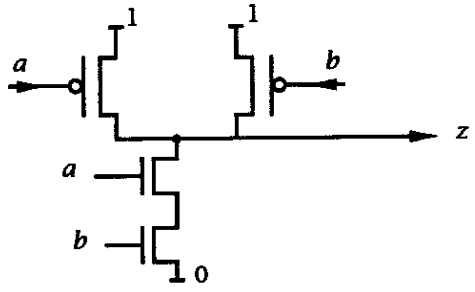
$$\begin{aligned} \neg x \underline{w} y \quad \equiv \quad & \neg x \mapsto y\uparrow \\ & x \mapsto y\downarrow . \end{aligned}$$

Most operators we use have more inputs than outputs. In general, however, the components we design have as many outputs as inputs. Hence, we need to reset the balance by introducing at least one operator, the *fork*, with more outputs than inputs. A fork with two outputs is defined as

$$\begin{aligned} x \underline{f}(y,z) \quad \equiv \quad & x \mapsto y\uparrow, z\uparrow \\ & \neg x \mapsto y\downarrow, z\downarrow . \end{aligned}$$

The wire and the fork are the only two operators that are implemented not as a pull-up/pull-down circuit —called a *restoring* circuit— but as a simple conducting interconnection between input and outputs.

Figure 3. CMOS implementation of a *nand*-gate.



5.2 *The Wire as a Renaming Operator*

Because the implementation of a wire is the same as that of a node, the wire behaves as a renaming operator when composed with another operator: The composition of an arbitrary operator O with output variable x with the wire $x \underline{w} y$ is equivalent to O in which x is renamed y. The composition of operator O with input variable x with the wire $y \underline{w} x$ is equivalent to O in which x is renamed y. (Observe that O can even be a wire.)

Unfortunately, the fork is not a renaming operator since the concurrent assignments to the different outputs of the fork are not completed simultaneously. In order to use a fork as a renaming operator, we will later have to make the timing assumption that such a fork is *isochronic*.

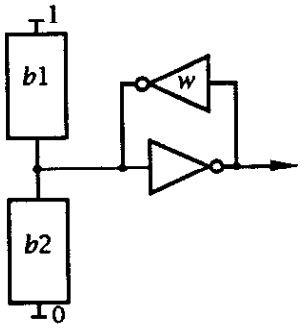
5.3 *Combinational Operators with Two Inputs*

We construct all functions B of two variables x and y such that

$$B \mapsto z\uparrow$$
$$\neg B \mapsto z\downarrow .$$

We get for B: $x \wedge y$, $x \vee y$, and $x = y$. We will not list the functions obtained by inverting inputs of B. (In the figures, a negated input or output is represented by a small circle on the corresponding line.) This gives the following set:

Figure 4. A static implementation of a state-holding operator.



The *and*, with the infix notation $(x, y) \triangle z$, is defined as

$$\begin{aligned} x \wedge y &\mapsto z\uparrow \\ \neg x \vee \neg y &\mapsto z\downarrow . \end{aligned}$$

The *or*, with the infix notation $(x, y) \underline{\vee} z$, is defined as

$$\begin{aligned} x \vee y &\mapsto z\uparrow \\ \neg x \wedge \neg y &\mapsto z\downarrow . \end{aligned}$$

The *equality*, with the infix notation $(x, y)\ \underline{eq}\ z$, is defined as

$$\begin{aligned} x = y &\mapsto z\uparrow \\ x \neq y &\mapsto z\downarrow . \end{aligned}$$

5.4 *State-Holding Operators with Two Inputs*

Next, we construct all different two-input-one-output operators of the form

$$\begin{aligned} b1 &\mapsto z\uparrow \\ b2 &\mapsto z\downarrow \end{aligned}$$

such that $\neg b1 \vee \neg b2$ holds at any time, but $b1 \neq \neg b2$. We select for $b1$ either $x \wedge y$, or $x \vee y$, or $x = y$. For each choice of $b1$, we construct $b2$ as any of the effective strengthenings of $\neg b1$.

For $b1 \equiv (x \wedge y)$, we get for $b2$: $\neg x \wedge \neg y$, $\neg x \wedge y$, $\neg x$, and $x \neq y$. The first three choices of $b2$ lead to the following state-holding operators:

The *C-element*:

$$\begin{aligned} (x, y)\ \underline{C}\ z \quad \equiv \quad x \wedge y &\mapsto z\uparrow \\ \neg x \wedge \neg y &\mapsto z\downarrow . \end{aligned}$$

(The C-element, introduced by David Muller, is described in [15].)

The *switch*:

$$\begin{aligned} (x, y)\ \underline{sw}\ z \quad \equiv \quad x \wedge y &\mapsto z\uparrow \\ \neg x \wedge y &\mapsto z\downarrow . \end{aligned}$$

The *asymmetric C-element*:

$$(x, y)\ \underline{aC}\ z \equiv \begin{array}{lll} x \wedge y & \mapsto & z\uparrow \\ \neg x & \mapsto & z\downarrow . \end{array}$$

For $b2 \equiv (x \neq y)$, we get the operator

$$\begin{array}{lll} x \wedge y & \mapsto & z\uparrow \\ x \neq y & \mapsto & z\downarrow . \end{array}$$

If the stability condition is fulfilled, however, this operator is not state-holding. Because of the stability requirement, the state in which $\neg x \wedge \neg y$ holds —the "storage state"— can be reached only from states $x \wedge \neg y$ and $\neg x \wedge y$. In both states, $\neg z$ holds, and, therefore, $\neg z$ holds in the storage state. Hence, we can weaken the guard of the second PR as $(x \neq y) \vee (\neg x \wedge \neg y)$, i.e., $\neg x \vee \neg y$. Hence, the operator is equivalent to the *and*-operator $(x, y)\ \underline{\wedge}\ z$.

For $b1 \equiv (x \vee y)$, no effective strengthening of $\neg b1$ is possible.

For $b1 \equiv (x = y)$, we get the operator:

$$\begin{array}{lll} x = y & \mapsto & z\uparrow \\ x \wedge \neg y & \mapsto & z\downarrow . \end{array}$$

If the stability condition is fulfilled, however, this operator is not state-holding for the same reasons that the operator with $b1 \equiv x \wedge y$ and $b2 \equiv (x \neq y)$ is not.

5.5 *Flip-Flop*

The canonical form we choose for the *flip-flop* is

$$(x, y)\ \underline{ff}\ z \equiv \begin{array}{lll} x & \mapsto & z\uparrow \\ \neg y & \mapsto & z\downarrow , \end{array}$$

which requires the invariance of $\neg x \vee y$ to satisfy noninterference. Observe that the flip-flop $(x, y)\ \underline{ff}\ z$ can always be replaced with the *C*-element $(x, y)\ \underline{C}\ z$, but not vice versa.

6 *Multi-Input Operators*

Since there are already 164 different operators with three inputs and one output, we shall not pursue the systematic enumeration that we started with two-input operators. We use *n*-input *and*, *or*, *C-element*, whose definitions are straightforward.

We use a *multi-input flip-flop* defined as

$$(x_1, \ldots, x_k, y_1, \ldots, y_l)\ \underline{mff}\ z \quad \equiv \quad \begin{array}{lll} \bigvee i : x_i & \mapsto & z\uparrow \\ \bigvee i : \neg y_i & \mapsto & z\downarrow \end{array}$$

where $(\forall i : \neg x_i) \vee (\forall i : y_i)$.

We also use the combinational *if*-operator —sometimes called *multiplexer*— defined as

$$(x, y, z)\ \underline{if}\ u \quad \equiv \quad \begin{array}{lll} (x \wedge y) \vee (\neg x \wedge z) & \mapsto & u\uparrow \\ (x \wedge \neg y) \vee (\neg x \wedge \neg z) & \mapsto & u\downarrow \ . \end{array}$$

The most general and most often used operator is the *generalized C-element*, of which all other forms of C-elements are a special case. It implements a pair of PRs

$$\begin{array}{lll} B1 & \mapsto & x\uparrow \\ B2 & \mapsto & x\downarrow \end{array}$$

in which $B1$ and $B2$ are arbitrary conjunctions of elementary terms. (As usual, the two guards have to be mutually exclusive.) For example,

$$\begin{array}{lll} a \wedge b \wedge \neg c & \mapsto & x\uparrow \\ \neg a \wedge d & \mapsto & x\downarrow \end{array}$$

can be directly implemented with a generalized C-element. Observe that the limiting factor for the size of the guards is not the number of inputs, but the number of terms in a conjunction.

7 *Arbiter and Synchronizer*

So far, we have considered only PR sets in which all guards are stable and noninterfering. But we shall have to implement sets of guarded commands —selections or repetitions— in which the guards are *not* mutually exclusive, as in the probe-selection example. Therefore, we need at least one operator that provides a nondeterministic choice between two **true** guards.

7.1 *Arbiter*

The simplest selection between nonexclusive guards is of the form

$$*[[x \rightarrow \cdots \;[]\; y \rightarrow \cdots \;]],$$

where x and y are simple boolean variables, and the two guards are stable. In order to distinguish among the three basic states of the system —i.e., neither x nor y is selected, x is selected, or y is selected— we must introduce two outputs, say u and v, as follows:

$$*[[x \rightarrow u\uparrow;\ \cdots \;[]\; y \rightarrow v\uparrow;\ \cdots \;]].$$

Initially, $\neg u \wedge \neg v$ holds as coding of the state "no selection made". Hence, when the selection is considered completed, which is just a matter of definition, u and v should be set back to **false**. We get

$$*[[x \rightarrow u\uparrow;\ [\neg x];\ u\downarrow \;[]\; y \rightarrow v\uparrow;\ [\neg y];\ v\downarrow \;]]. \tag{6}$$

If $\neg u \wedge \neg v$ holds initially, $\neg u \vee \neg v$ holds at any time.

The preceding program is a description of the operator known as the "basic arbiter" or "mutual-exclusion element," denoted as (x, y) $\underline{arb}$ (u, v). Observe that the choice between the two guards is not fair.

7.2 *Synchronizer*

When negated probes are used, for instance to implement fairness, we have to implement selection commands with unstable guards. The synchronizer is the only operator that accepts nonstable guards. It is defined as

$$*[[b \wedge z \rightarrow u\uparrow;\ [\neg z];\ u\downarrow \;[]\; \neg b \wedge z \rightarrow v\uparrow;\ [\neg z];\ v\downarrow \;]]. \tag{7}$$

Variable b may change at any time from **false** to **true**, but both b and z remain **true** until u or v has changed. Hence, the guard $\neg b \wedge z$ is unstable, whereas the guard $b \wedge z$ is stable. As in the arbiter case, if $\neg u \wedge \neg v$ holds initially, $\neg u \vee \neg v$ holds at any time. (The synchronizer operator was introduced in [7].)

7.3 *Implementation and Metastability*

The PR sets for (6) and (7) necessarily contain unstable rules. The PR set for the "unstable arbiter" is

$$\begin{aligned} x \wedge \neg v &\mapsto u\uparrow \\ y \wedge \neg u &\mapsto v\uparrow \\ \neg x \vee v &\mapsto u\downarrow \\ \neg y \vee u &\mapsto v\downarrow . \end{aligned}$$

The PR set for the "unstable synchronizer" is

$$\begin{aligned} b \wedge z \wedge \neg v &\mapsto u\uparrow \\ \neg b \wedge z \wedge \neg u &\mapsto v\uparrow \\ \neg z \vee v &\mapsto u\downarrow \\ \neg z \vee u &\mapsto v\downarrow . \end{aligned}$$

The first two PRs of the arbiter are unstable and can fire concurrently. The same holds for the first two production rules of the synchronizer: Since b can change from **false** to **true** at any time, both guards may evaluate to **true**.

Let us analyze the PR set implementation of the arbiter. The synchronizer case is very similar. The state $x \wedge y \wedge (u = v)$ of the arbiter is called *metastable.* When started in the metastable state, with $\neg u \wedge \neg v$, the set of PRs specifying the arbiter may produce the following unbounded sequence of firings:

$$*[(u\uparrow, v\uparrow); (u\downarrow, v\downarrow)].$$

In the implementation, nodes u and v may stabilize to a common intermediate voltage value for an unbounded period of time. Eventually, the inherent asymmetry of the physical realization (impurities, fabrication flaws, thermal noise, etc.) will force the system into one of the two stable states where $u \neq v$. But there is no upper bound on the time the metastable state will last, which means that it is impossible to include an arbitration device into a clocked system with absolute certainty that a timing failure cannot occur.

The spurious values of u and v produced during the metastable state must be eliminated since they violate the requirement $\neg u \vee \neg v$. Hence, we compose

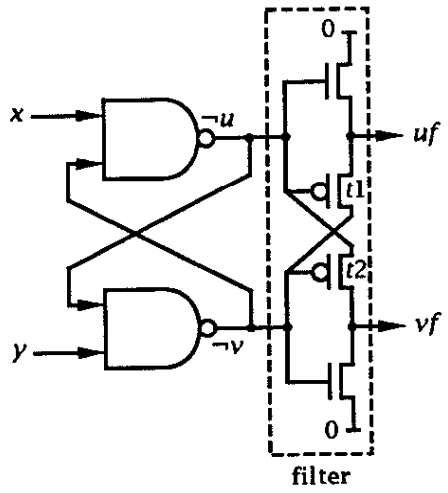
the "bare" arbiter with a "filter" taking u and v as input and producing uf and vf as "filtered outputs". The net effect of the filter is

$$uf, vf := (u \wedge \neg v), (v \wedge \neg u).$$

(In the CMOS construction of the filter shown in Figure 5, we use the threshold voltages to our advantage: The channel of transistor $t1$ is conducting only when $(u \wedge \neg v)$ holds, and the channel of transistor $t2$ is conducting only when $(v \wedge \neg u)$ holds.)

In delay-insensitive design, the correct functioning of a circuit containing an arbiter or a synchronizer is independent of the duration of the metastable state; therefore, relatively simple implementations of arbiters and synchronizers can be used. In synchronous design, however, the implementations have to meet the additional constraint that the probability of the metastable state lasting longer than the clock period should be negligible.

Figure 5. An implementation of the basic arbiter.



8 *Sequencing and Stability*

In the second part of this chapter, we shall see how an arbitrary program in the source notation can be decomposed —by a transformation called *handshaking expansion*— into a collection of sequences of the type

$$S \equiv *[[w_0];\ t_0;\ [w_1];\ t_1; \ldots ; [w_{n-1}];\ t_{n-1}] .$$

The w_i, the *wait-conditions*, are boolean expressions, possibly identical to **true**, and the t_i are simple assignments. The extension to the case of multiple assignments between the wait-conditions is straightforward.

The next step of the compilation procedure —the *production-rule expansion*— (also to be explained in the second part) is the transformation of S into a semantically equivalent set of production rules. Let

$$P \equiv \{b_i \mapsto t_i | 0 \leq i < n\}$$

be such a set.

Notations and Definitions For an arbitrary PR p, $p.g$ and $p.a$ denote the guard and the assignment of p, respectively. The predicate $R(a)$, the *result* of the simple assignment a, is defined as: $R(x\uparrow) = x$, and $R(x\downarrow) = \neg x$. An execution of a PR that changes the value of the assigned variable is called *effective*; otherwise, it is called *vacuous*.

With these definitions, the stability of a PR can be reformulated as follows:

Stability A PR p is stable in a computation if and only if $p.g$ can be falsified only in states where $R(p.a)$ holds.

The production-rule expansion algorithm compiles a handshaking expansion S into a set P of PRs, all of which are stable except those whose guards contain negated probes. Since, as we shall see, the guards of the PRs are obtained by strengthening the wait-conditions of S, the stability of the wait-conditions is necessary to satisfy the stability of the PRs.

A wait-condition w is stable if once w is **true**, it remains **true** at least until the completion of the following assignment. Unstable wait-conditions can be caused by negated probes only. These cases are dealt with separately by introducing synchronizers. (An example of how this is achieved is given in Section 22.)

8.1 *Sequencing*

The set P of PRs implements S when the following conditions are fulfilled:

1. *Guard strengthening:* The guards of the PRs of P are obtained by strengthening the wait conditions of S: $\forall i :: b_i \Rightarrow w_i$ and, in the initial state, $w_0 \Rightarrow b_0$.
2. *Sequential execution:* $(\mathbf{N}i :: b_i \wedge \neg R(t_i)) \leq 1$, i.e., at most one effective PR can be executed at a time.
3. *Program-order execution:* The order of execution of effective PRs of P is the order specified by S, called the *program order*, and no deadlock is introduced in the construction of P.

As we shall see in Part 2, it is not always possible to construct, for a given handshaking expansion, a PR set that satisfies the preceding three conditions. In certain cases, the handshaking expansion must be augmented with assignments to new variables, called *state variables*. This transformation, which is always possible, will be explained in Part 2.

8.2 *Acknowledgment*

Fulfilling the second and third conditions requires that for any two PRs $p : b \mapsto t$ and $p' : b' \mapsto t'$, such that p immediately precedes p' in the program order,

$$b' \Rightarrow R(t)$$

holds in the states where p' is effectively executed. We say that b' is the *acknowledgment* of t. Hence the following property:

Acknowledgment Property For a PR set executed in program order, the guard of each PR is an acknowledgment of the immediately preceding assignment.

We shall see that the acknowledgment property is necessary but not sufficient to ensure program-order execution.

We use two kinds of acknowledgments, depending on the type of variable used in the assignment. But other forms of acknowledgments can be envisioned. If t assigns an internal variable, then the acknowledgment is implemented by strengthening b' as $b' \wedge R(t)$.

For example, if t is $x\uparrow$, the acknowledgment is $b' \wedge x$.

If t assigns an external variable, i.e., a variable that implements a communication command, another kind of acknowledgment, which we shall introduce later, can be used. For instance, if *lo* is an output variable used together with input variable *li* to implement a so-called active handshaking protocol, a possible acknowledgment of $lo\uparrow$ is *li*, since $li \Rightarrow lo$ at this point of the protocol.

8.3 *Implementation of Stability*

Consider a PR set P, which implements a given program S. We are going to show that the acknowledgment property, which is necessary to construct a P that implements S, is also sufficient to guarantee stability.

The execution of a PR p of P establishes a path between a constant node (either *VDD* or *GND*), and the node implementing the variable —say, x— assigned by p. Either $p.g$ holds forever after p, or the firing of another PR I, the *invalidating* PR of p, will establish $\neg p.g$, thereby cutting the path from the constant node to x.

Let $\tilde{p}$ be the complementary PR of p, i.e., the PR with the complementary assignment. If the PR set contains both p and $\tilde{p}$, then it also contains I because of the noninterference requirement between complementary PRs. And we have the order of execution:

$$p \preceq I \prec \tilde{p}\,.$$

In all the states between I and $\tilde{p}$, the original path to x is cut. In that case, we have to see to it that the assignment to x is completed before the path is cut. Hence the following requirement:

Completion requirement Assignment $p.a$ is completed when a PR q is completed whose guard is an acknowledgment of $p.a$. The execution order of the PR set must satisfy

$$p \prec q \preceq I\,.$$

Since this requirement is already implied by the acknowledgment property, the construction of P automatically guarantees stability.

8.4 *Self-Invalidating PRs*

Definition A PR p is *self-invalidating* when $R(p.a) \Rightarrow \neg p.g$.

For example, $\neg x \mapsto x\uparrow$ is self-invalidating.

Self-invalidating PRs are excluded by the completion requirement since it implies $I \neq p$.

For instance, the circuit consisting of an inverter with its output connected to its input is excluded by the completion requirement since it corresponds

to the PR set:

$$\neg x \mapsto x\uparrow$$
$$x \mapsto x\downarrow$$

and the two PRs of the set are self-invalidating. However, the PR set

$$\neg x \mapsto y\uparrow$$
$$y \mapsto x\uparrow$$
$$x \mapsto y\downarrow$$
$$\neg y \mapsto x\downarrow$$

fulfills the completion requirement, although it is the same circuit as previously, since the only change is the addition of the wire $y \underline{w} x$.

We eliminate such "disguised" self-invalidating PRs by adding the following requirement:

Restoring Acknowledgment Requirement There is at least one restoring PR r satisfying $p \prec r \preceq I$, where r is restoring if it is not part of a wire or a fork.

With this extra requirement, all forms of self-invalidating PRs are eliminated.

It is remarkable that the acknowledgment requirement, which is necessary to enforce the sequential execution of a PR set, is also sufficient to satisfy stability. From now on, we can manipulate PRs as if the transitions were discrete. We have, however, made no simplifying assumption on the physical behavior of the system. The only physical requirement so far is that of monotonicity.

Another requirement on the implementation is that the rings of operators that constitute a circuit keep oscillating. It turns out that eliminating self-invalidating PRs enforces the condition that a ring contain at least three restoring operators, which is a necessary (and in practice also sufficient) condition for the ring to oscillate, thanks to the "gain" property of restoring gates. (See [14] for an explanation of gain.)

Part II: The Compilation Method

In this part, we describe how a program in the source notation is transformed into a semantically equivalent set of VLSI operators. Four major trans-

an intermediate program representation, between communicating processes and PRs, that allows for important algebraic manipulations of the program: reshuffling, process factorization, and process quotient. We illustrate the method with a series of examples that covers practically all cases.

9 *Process Decomposition*

The first step of the compilation, called *process decomposition*, consists in replacing one process with several processes by application of the following rule:

Decomposition Rule A process P containing an arbitrary program part S is semantically equivalent to two processes, $P1$ and $P2$, where $P1$ is derived from P by replacing S with a communication action, C, on a newly introduced channel (C, D) between $P1$ and $P2$, and $P2$ is the process $*[[\overline{D} \rightarrow S; D]]$.

The structure of $P2$ will be used so frequently that we introduce an operator to denote it: the *call* operator. We denote it by (D/S), and we say that D *calls* (or *activates*) S.

Observe that process decomposition does not introduce concurrency. Although $P1$ and $P2$ are potentially concurrent, they are never active concurrently; $P2$ is activated from $P1$, much as a procedure or a coroutine would be. The newly created subprocesses may share variables, but, since the subprocesses are never active concurrently, there is no conflicting access to the shared variables. The subprocesses may also share channels; this will require a special implementation for such channels. Decomposition is applied for each construct of the language. For construct S, the corresponding process $P2$ can be simplified as follows:

If S is the selection $[B_1 \rightarrow S_1 [] B_2 \rightarrow S_2]$, $P2$ is simplified as

$$
\begin{array}{l}
*[[\overline{D} \wedge B_1 \rightarrow S_1; D \\
\quad [] \, \overline{D} \wedge B_2 \rightarrow S_2; D \\
\quad]] .
\end{array}
\tag{8}
$$

If S is the repetition $*[B_1 \rightarrow S_1 [] B_2 \rightarrow S_2]$, $P2$ is simplified as

$$
\begin{aligned}
&*[[\overline{D} \wedge B_1 \rightarrow S_1 \\
&\quad [] \overline{D} \wedge B_2 \rightarrow S_2 \\
&\quad [] \overline{D} \wedge \neg B_1 \wedge \neg B_2 \rightarrow D \\
&\quad]] .
\end{aligned}
\tag{9}
$$

The assignment $x := B$, where B is an arbitrary boolean expression, is implemented as the selection $[B \rightarrow x\uparrow [] \neg B \rightarrow x\downarrow]$, which gives for $P2$

$$
\begin{aligned}
&*[[\overline{D} \wedge B \rightarrow x\uparrow; D \\
&\quad [] \overline{D} \wedge \neg B \rightarrow x\downarrow; D \\
&\quad]] .
\end{aligned}
$$

The generalizations to the cases of an arbitrary number of guarded commands in selection and repetition are obvious. All assignments to the same variable are also grouped in the same process. Process decomposition is applied repeatedly until the right-hand side of each guarded command is a straight-line program.

Process decomposition makes it possible to reduce a process with an arbitrary control structure to a set of subprocesses of only two different types: either a (finite or infinite) sequence of communication actions, or a repetition of type (8) or (9).

10 *Handshaking Expansion*

The next step of the transformation, the *handshaking expansion*, replaces each communication action in a program with its implementation in terms of elementary actions, and each channel with a pair of wire operators. We shall first ignore the issue of message transmission and implement only the synchronization property of communication primitives.

Channel (X, Y) is implemented by the two wires (xo $\underline{w}$ yi) and (yo $\underline{w}$ xi). If X belongs to process $P1$ and Y to process $P2$, then xo and xi belong to $P1$, and yo and yi to $P2$. Initially, xo, xi, yo, and yi —which we will call the "handshaking variables of (X, Y)"— are **false**. Assume that the program has been proven to be deadlock-free and that we can identify a pair of matching actions X and Y in $P1$ and $P2$, respectively. We replace X and Y by the sequences U_x and U_y,

respectively, where

$$U_x \equiv xo\uparrow;\ [xi] \tag{10}$$
$$U_y \equiv [yi];\ yo\uparrow\ .$$

Also,

$$\begin{aligned} xo &\mapsto yi\uparrow \\ \neg xo &\mapsto yi\downarrow \\ yo &\mapsto xi\uparrow \\ \neg yo &\mapsto xi\downarrow, \end{aligned} \tag{11}$$

by definition of the wires. By (10) and (11), any concurrent execution of $P1$ and $P2$ contains the following sequence of assignments:

$$xo\uparrow;\ yi\uparrow;\ yo\uparrow;\ xi\uparrow\ .$$

10.1 *Simultaneous Completion of Nonatomic Actions*

We introduce a definition of *completion* of a nonatomic action which makes it possible to use the notion of simultaneous completion of two nonatomic actions.

By definition, the execution of an atomic action is considered instantaneous, and thus the simultaneous completion of two atomic actions does not make sense. (Atomic actions are simple assignments $x\uparrow$ and $x\downarrow$, and evaluation of simple guards, i.e., guards containing one variable. A wait action of the form $[ai]$ is a nonatomic action that may be treated as the repetition $*[\neg ai \rightarrow skip]$.)

A nonatomic action is *initiated* when its first atomic action is executed. A nonatomic action is *terminated* when its last atomic action is executed.

For nonatomic actions, the notion of completion does not coincide with that of termination. A nonatomic action might be considered completed even if it has not terminated, i.e., even if some atomic actions that are part of the action have not been executed. The definition of suspension is derived from that of completion.

Definition A nonatomic action X is completed when it is initiated and is guaranteed to terminate, i.e., when all possible continuations of the computation contain the complete sequence of atomic actions of X.

The preceding definition can be further explained as follows: Consider a prefix $t1$ of an arbitrary *trace* of a computation. (A trace is a sequence of

atomic actions corresponding to a possible execution of the program.) The completion of X is identified with the point in the computation where $t1$ has been completed, if (1) X is initiated in $t1$, and (2) all possible sequences $t2$, such that $t1$ extended with $t2$ is a valid trace of the computation, contain the remaining atomic actions of X. *Hence the completions of two nonatomic actions coincide if their completion points coincide.*

(Observe that there may be several points in a trace that can act as completion point, which makes it easier to align the two completion points of two overlapping sequences so as to implement the bullet operator.)

Definition Between initiation and completion, an action is *suspended.*

These definitions of completion and suspension are valid because they satisfy the three semantic properties of completion and suspension that are used in correctness arguments, namely:

1. $\{\mathbf{c}X = x\}\ X\ \{\mathbf{c}X = x + 1\}$,
2. $\mathbf{q}X \Rightarrow pre(X)$, where $pre(X)$ is any precondition of X in terms of the program variables and auxiliary program variables,
3. If X is completed, eventually X is terminated.

These definitions will be used to implement the bullet operator and the communication primitives as defined by axioms $A1$ and $A2$. Consider the interleaving of U_x and U_y. At the first semicolon, i.e., after $xo\uparrow$, U_x has been initiated, but it cannot be considered completed since the valid continuation that does not contain U_y does not contain the rest of U_x. At the second semicolon, both U_x and U_y have been initiated, and thus all continuations contain the rest of the interleaving of U_x and U_y. Hence, U_x and U_y are guaranteed to terminate when they are both initiated, i.e., they fulfill $A1$ and $A2$.

10.2 *Four-Phase Handshaking*

Unfortunately, when the communication implemented by U_x and U_y terminates, all handshaking variables are **true**. Hence, we cannot implement the next communication on channel (X, Y) with U_x and U_y. The complementary implementation, however, can be used for the next matching pair, that is:

$$D_x \equiv xo\downarrow;\ [\neg xi]$$

$$D_y \equiv [\neg yi];\ yo\downarrow\ .$$

The solution consisting in alternating U_x and D_x as an implementation of X, and U_y and D_y as an implementation of Y, is called *two-phase handshaking,*

or *two-cycle signaling*. Since it is in most cases impossible to determine syntactically which X- or Y-actions follow each other in an execution, the general two-phase handshaking implementations require testing the current value of the variables as follows:

$$xo := \neg xo;\ [xi = xo]$$

$$[yi \neq yo];\ yo := \neg yo\ .$$

In general, we prefer to use a simpler solution, known as *four-phase handshaking*, or *four-cycle signaling*. In a four-phase handshaking protocol, X-actions are implemented as "$U_x; D_x$" and Y-actions as "$U_y; D_y$". Observe that the D-parts in X and Y introduce an extra communication between the two processes whose only purpose is to reset all variables to **false**.

Both protocols have the property that for a matching pair (X, Y) of actions, the implementation is not symmetrical in X and Y. One action is called *active* and the other one *passive*. The four-phase implementation, with X active and Y passive, is

$$X \equiv xo\uparrow;\ [xi];\ xo\downarrow;\ [\neg xi] \tag{12}$$

$$Y \equiv [yi];\ yo\uparrow;\ [\neg yi];\ yo\downarrow\ . \tag{13}$$

(Later, we will introduce an alternative form of active implementation, called *lazy-active*.) Although four-phase handshaking contains twice as many actions as two-phase handshaking, the actions involved are simpler and are more amenable to the algebraic manipulations we shall introduce later. When operator delays dominate the communication costs, which is the case for communication inside a chip, four-phase handshaking will, in general, lead to more efficient solutions. When transmission delays dominate the communication costs, which is the case for communication between chips, two-phase handshaking is preferred.

10.3 *Probe*

A simple implementation of the probe $\overline{X}$ is xi, with X implemented as passive. (Given our definition of suspension, the proof that this implementation of the probe fulfills its definition is straightforward.)

A probed communication action $\overline{X} \rightarrow \ldots X$ is then implemented as

$$xi \rightarrow \ldots xo\uparrow;\ [\neg xi];\ xo\downarrow\ .$$

10.4 *Choice of Active versus Passive Implementation*

When no action of a matching pair is probed, the choice of which action should be active and which passive is arbitrary, but a choice has to be made. The choice can be important for the composition of identical circuits. A simple rule is that, for a given channel (X, Y), all actions on one port (called the *active port*) are active, and all actions on the other port (called the *passive port*) are passive. If $\overline{X}$ is used, all X-actions are passive— with the obvious restriction that $\overline{Y}$ cannot be used in the same program.

We shall see, however, that this criterion for choosing active and passive ports may conflict with another criterion related to the implementation of input and output commands.

10.5 *Properties of the Handshaking Protocol*

For a matching pair (X, Y) of actions implemented as (12) and (13), and the wires $(xo \underline{w} yi)$ and $(yo \underline{w} xi)$, the concurrent execution of X and Y causes the sequence of assignments

$$xo\uparrow;\ yi\uparrow;\ yo\uparrow;\ xi\uparrow;\ xo\downarrow;\ yi\downarrow;\ yo\downarrow;\ xi\downarrow,$$

called the *handshaking protocol*. The following properties of the handshaking protocol play an important role in the compilation method.

Property 1 For xo and xi used as in the active protocol of (12), xi is an acknowledgment of $xo\uparrow$ and $\neg xi$ is an acknowledgment of $xo\downarrow$. For yo and yi used as in the passive protocol of (13), $\neg yi$ is an acknowledgment of $yo\uparrow$ and yi is an acknowledgment of $yo\downarrow$.

Property 2 In (12) and (13), D_x and D_y are used only to reset all variables to **false**. Hence, provided that the cyclic order of the actions of (12) and (13) is maintained, the sequences D_x and D_y can be inserted at any place in the program of each of the processes without invalidating the semantics of the communication involved. This transformation, called *reshuffling*, may introduce a deadlock.

Property 3 The wait-actions of (12) and (13) are stable. Reshuffling maintains the stability.

Reshuffling, which is the source of significant optimizations, will be used extensively. It is therefore important to know when Property 2 can be applied without introducing deadlock.

There are two simple cases where the reshuffling of sequence "$U_x;D_x;S$" into sequence "$U_x;S;D_x$" does not introduce deadlock:

S contains no communication action, or

X is an internal channel introduced by process decomposition.

11 *Production-Rule Expansion*

Production-rule expansion is the transformation from a handshaking expansion to a set of PRs. It is the most crucial and most difficult step of the compilation since it requires the enforcement of sequencing by semantic means. It consists of three steps:

1. State assignment,
2. Guard strengthening,
3. Symmetrization.

We shall explain the algorithms for production-rule expansion with an example: the implementation of the simple process (L/R), where R is an active channel. This process is one of the basic building blocks for implementing sequencing. The handshaking expansion gives

$$*\,[[li];\ ro\uparrow;\ [ri];\ ro\downarrow;\ [\neg ri];\ lo\uparrow;\ [\neg li];\ lo\downarrow]\,. \tag{14}$$

We now consider the handshaking expansion as the specification of the implementation: Any implementation of the program has to satisfy the ordering defined by (14). The next step is to construct a production-rule set that satisfies this ordering. We start with the production-rule set that is syntactically derived from (14):

$$\begin{aligned} li &\mapsto ro\uparrow \\ ri &\mapsto ro\downarrow \\ \neg ri &\mapsto lo\uparrow \\ \neg li &\mapsto lo\downarrow\,. \end{aligned}$$

(As a clue to the reader, PRs of a set are listed in program order.)

Since the program is deadlock-free, effective execution of the PRs in program order is always possible. Some other execution orders, however, may also be possible. The production-rule set satisfies the handshaking-expansion specification if, and only if, the only possible execution order is the program order. If execution orders other than the program order are possible for the

production-rule set, the guards of some rules are strengthened so as to eliminate these execution orders.

In our example, program order is not the only execution order for the syntactic production-rule set: Since $\neg ri$ holds initially, the third PR can be executed first. This is also true for the fourth PR, but the execution of the fourth rule in the initial state is vacuous. Because all handshaking variables of R are back to **false** when R is completed, we cannot find a guard for the transition $lo\uparrow$ that holds only as a precondition of $lo\uparrow$ in (14). Hence, we cannot distinguish the state following R from the state preceding R, and thus the sequential execution condition introduced in Section 8 cannot be satisfied.

This is a general problem, since it arises for each unshuffled communication action. In order to fulfill the sequential-execution condition, we have to guarantee that each state of the handshaking expansion is unique, i.e., that there exists a predicate in terms of variables of the program that holds only in this state. The task of transforming the handshaking expansion so as to make each state unique is called *state assignment*.

11.1 *State Assignment with State Variables*

The first technique to define uniquely the state in which the transition $lo\uparrow$ is to take place consists in introducing a state variable, say x, initially **false**. Handshaking expansion (14) becomes

$$*\,[[li];\ ro\uparrow;\ [ri];\ x\uparrow;\ [x];\ ro\downarrow;\ [\neg ri];\ lo\uparrow;\ [\neg li];\ x\downarrow;\ [\neg x];\ lo\downarrow]\,. \tag{15}$$

Observe that (15) is semantically equivalent to (14) since the two sequences of actions that are added to (14), namely, $x\uparrow;[x]$ and $x\downarrow;[\neg x]$, are equivalent to a **skip**. (The newly introduced variable x is used nowhere else.)

There are several places where the two assignments to the state variable can be introduced. In general, a good heuristic is to introduce those assignments at such places that the alternation between waits and assignments is maintained. There are other heuristics, however, that can play a role in the placement of the variables.

Once state variables have been introduced so as to distinguish any two states of the handshaking expansion, it is possible to strengthen the guards of the PRs to enforce program-order execution. The basic algorithm for guard strengthening can be found in [10]. We shall not describe it here. Applied to

(15), it gives

$$\neg x \wedge li \;\mapsto\; ro\uparrow \tag{16}$$

$$ri \;\mapsto\; x\uparrow \tag{17}$$

$$x \;\mapsto\; ro\downarrow \tag{18}$$

$$x \wedge \neg ri \;\mapsto\; lo\uparrow \tag{19}$$

$$\neg li \;\mapsto\; x\downarrow \tag{20}$$

$$\neg x \;\mapsto\; lo\downarrow. \tag{21}$$

It is easy to check that the acknowledgment property is fulfilled and that the only possible execution order for the preceding production-rule set is the program order defined by (15).

12 *Operator Reduction*

The last step of the compilation, called *operator reduction*, groups together the PRs that assign the same variables. Those PRs are then identified with (and implemented as) an operator. The program is thus identified with a set of operators.

Since we have enforced the stability of each rule and noninterference between any two complementary rules, we can implement any set of PRs directly. (For reasons of efficiency, we must see to it that the guards do not contain too many variables in a conjunct, which would lead to too many transistors in series. Hence, the implementation of the set may also involve decomposing a PR into several PRs by introducing new internal variables.)

The direct implementation of the PR set (16) through (21) is straightforward:

(16) and (18) correspond to the asymmetric C-element $(\neg x, li)$ $\underline{aC}$ ro.

(19) and (21) correspond to the asymmetric C-element $(x, \neg ri)$ $\underline{aC}$ lo.

(17) and (20) correspond to the flip-flop (ri, li) $\underline{ff}$ x.

If the preceding operators are implemented as dynamic, this implementation of process (L/R) is the simplest possible. If static implementations of the operators are required, another implementation might be considered with fewer state-holding elements since, as we have explained in the first part, static state-holding operators are slightly more difficult to realize than combinational operators.

A last transformation, called *symmetrization*, may be performed on the PR set to minimize the number of state-holding operators. Since symmetrization also introduces inefficiencies of its own, however, it should not be applied blindly.

13 *Symmetrization*

Symmetrization is performed on the two guards of PRs $b1 \mapsto z\uparrow$ and $b2 \mapsto z\downarrow$, when one of the two guards, say, $b1$, is already in the form $x \wedge \neg b2$. If we replace guard $b2$ with $\neg x \vee b2$, then the two guards are complements of each other, i.e., the operator is combinational. Of course, weakening guard $b2$ is a dangerous transformation since it may introduce a new state where the guard holds. We have to check that this does not occur by checking the following invariant:

Given the new rule $\neg x \vee b2 \mapsto z\downarrow$, $\neg z$ must hold in any state where $\neg x \wedge \neg b2$ holds, i.e., we have to check the invariant truth of

$$x \vee b2 \vee \neg z\,.$$

13.1 *Operator Reduction of the (L/R)-element*

The symmetrization of PRs (16) and (18), and of (19) and (21) of the (*L*/*R*)-element, gives

$$\neg x \wedge li \mapsto ro\uparrow \tag{16}$$

$$ri \mapsto x\uparrow \tag{17}$$

$$\neg li \vee x \mapsto ro\downarrow \tag{18}$$

$$x \wedge \neg ri \mapsto lo\uparrow \tag{19}$$

$$\neg li \mapsto x\downarrow \tag{20}$$

$$ri \vee \neg x \mapsto lo\downarrow\,. \tag{21}$$

(16) and (18) correspond to the *and*-operator $(\neg x, li) \triangle ro$.

(17) and (20) correspond to the flip-flop $(ri, li)\ \underline{ff}\,x$.

(19) and (21) correspond to the *and*-operator $(x, \neg ri) \triangle lo$.

(17) and (20) can also be implemented as the C-element $(li, ri)\ \underline{C}\,x$.

The resulting circuit is shown in Figure 6. (The dot identifies the input that is activated first.) This implementation of (*L*/*R*), either with a flip-flop or with a C-element, is called a *Q-element*. The Q-element implementing (*L*/*R*) as before is described by the infix notation $(li, lo)\ \underline{Q}\,(ri, ro)$.

14 *Isochronic Forks*

In the previous operator reduction, li is an input to the flip-flop $(li, ri)\ \underline{ff}\,x$ and to the *and*-operator $(li, \neg x) \triangle ro$. Formally, in order to compose the PRs

together to form a circuit, we have to introduce the fork $li\,\underline{f}(l1, l2)$ and replace li by $l1$ as input of the *and*-operator, and by $l2$ as input of the flip-flop. We also have to introduce the forks $ri\,\underline{f}(r1, r2)$ and $x\,\underline{f}(x1, x2)$ for the same reason.

Let us analyze the effect of the first fork only. The PR set that includes the PRs of the fork is

$$li \mapsto l1\uparrow, l2\uparrow \tag{16a}$$

$$\neg x \wedge l1 \mapsto ro\uparrow \tag{16b}$$

$$ri \mapsto x\uparrow \tag{17}$$

$$\neg l1 \vee x \mapsto ro\downarrow \tag{18}$$

$$x \wedge \neg ri \mapsto lo\uparrow \tag{19}$$

$$\neg li \mapsto l1\downarrow, l2\downarrow \tag{20a}$$

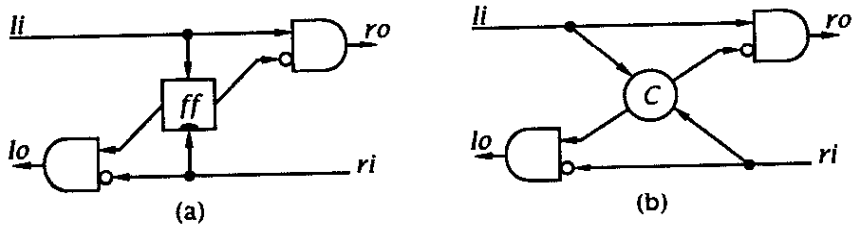
$$\neg l2 \mapsto x\downarrow \tag{20b}$$

$$ri \vee \neg x \mapsto lo\downarrow . \tag{21}$$

Now we observe that transition $l1\uparrow$ of (16a) is acknowledged by the guard of (16b) but $l2\uparrow$ is not, and transition $l2\downarrow$ of (20a) is acknowledged by the guard of (20b) but $l1\downarrow$ is not. Hence, the assignments $l2\uparrow$ and $l1\downarrow$ do not fulfill the completion requirement and thus are not stable!

We solve this problem by making a simplifying assumption: We assume that the fork is *isochronic*. That is, the difference in delays between the two branches of the fork is shorter than the delays in the operators to which the fork is an input. Hence, when a transition on one output is acknowledged and

Figure 6. Implementation of (L/R) with a Q-element.



thus completed, the transition on the other output is also acknowledged and thus completed.

This is the only timing condition that must be fulfilled. In general, the constraint is easy to meet because it is one-sided. The isochronicity requirement is more difficult to meet, however, when a negated input introduces an inverter on a branch of the fork, since the transition delays of an inverter are of the same order of magnitude as the transition delays of other operators. We have proved that, for the implementation of each language construct, these inverters can always be eliminated from the isochronic forks by simple transformations.[3] (See [1, 2].)

In [11], we have proved that the class of entirely delay-insensitive circuits is very limited: Practically all circuits of interest fall outside the class. We believe that the notion of isochronic fork is the weakest compromise to delay-insensitivity sufficient to implement any circuit of interest.

Which forks have to be isochronic is easy to decide by a simple analysis of the PR sets. For instance, the fork $ri\,\underline{f}(r1, r2)$ also has to be isochronic, but the fork $x\,\underline{f}(x1, x2)$ does not. We shall ignore the issue of isochronic forks in the rest of this presentation.

15 *Reshuffled Implementations of (L/R)*

We illustrate the use of reshuffling by deriving two other implementations of (L/R). If L is an internal channel introduced for process decomposition, we can reshuffle the handshaking expansions of L and R without the risk of introducing deadlock. Let us return to handshaking expansion (14).

15.1 *First Reshuffling*

We postpone the second half of the handshaking expansion of R —i.e., the sequence $ro\downarrow; [\neg ri]$— until after $[\neg li]$. We get

$$*[[li];\ ro\uparrow;\ [ri];\ lo\uparrow;\ [\neg li];\ ro\downarrow;\ [\neg ri];\ lo\downarrow]\,.$$

The syntactic PR expansion we now derive is already "program-ordered":

$$\begin{aligned} li &\mapsto ro\uparrow \\ ri &\mapsto lo\uparrow \\ \neg li &\mapsto ro\downarrow \\ \neg ri &\mapsto lo\downarrow\,. \end{aligned}$$

3. These transformations have not been applied to the circuits presented here as examples, but they are always applied before the circuits are actually implemented.

The first and third rules specify the wire $(li \underline{w} ro)$; the second and fourth rules specify the wire $(ri \underline{w} lo)$. Hence, the implementation reduces to two wires!

15.2 *Second Reshuffling: The D-element*

We now postpone the whole handshaking expansion of R until after $[\neg li]$. We get

$$*[[li]; lo\uparrow; [\neg li]; ro\uparrow; [ri]; ro\downarrow; [\neg ri]; lo\downarrow] .$$

We need to introduce a state variable, say x, as follows:

$$*[[li]; x\uparrow; [x]; lo\uparrow; [\neg li]; ro\uparrow; [ri]; x\downarrow; [\neg x]; ro\downarrow; [\neg ri]; lo\downarrow] .$$

The PR expansion gives

$$\begin{aligned} li &\mapsto x\uparrow \\ (ri \vee) x &\mapsto lo\uparrow \\ x \wedge \neg li &\mapsto ro\uparrow \\ ri &\mapsto x\downarrow \\ (li \vee) \neg x &\mapsto ro\downarrow \\ \neg x \wedge \neg ri &\mapsto lo\downarrow . \end{aligned}$$

The terms between parentheses have been added for symmetrization. The operator reduction gives

$$\begin{aligned} (li, \neg ri) &\ \underline{ff}\ x \\ (ri, x) &\ \underline{\vee}\ lo \\ (x, \neg li) &\ \underline{\wedge}\ ro . \end{aligned}$$

The flip-flop can be replaced with the C-element $(li, \neg ri) \underline{C} x$. The circuit, shown in Figure 7, is called a D-element.

16 *Sequencing*

There are many ways to implement the sequencing of n arbitrary actions. We shall introduce the basic operators that are used in the most straightforward implementations.

16.1 *The Active-Active Buffer*

Consider the program $*[S_1;S_2]$, where S_1 and S_2 are two arbitrary program parts. Process decomposition of this program gives

$$*[L;R] \,\|\, (L'/S_1) \,\|\, (R'/S_2)\,.$$

Hence the basic sequencing operator is the process

$$B(L_a,R_a) \equiv *[L;R]\,,$$

where both L and R are active. This process is called an *active-active buffer.* The handshaking expansion gives

$$*\,[lo\uparrow;\ [li];\ lo\downarrow;[\neg li];\ ro\uparrow;\ [ri];\ ro\downarrow;\ [\neg ri]]\,. \tag{22}$$

Since ri is **false** initially, we can rewrite (22) as

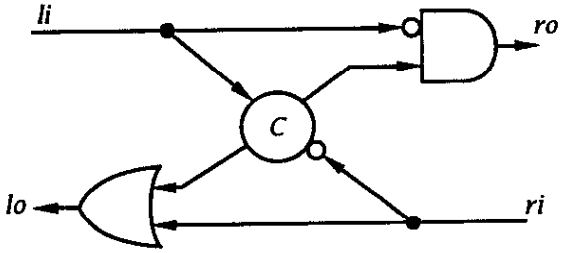
$$*\,[[\neg ri];\ lo\uparrow;\ [li];\ lo\downarrow;[\neg li];\ ro\uparrow;\ [ri];\ ro\downarrow]\,. \tag{23}$$

By comparing (23) with (14) —the handshaking expansion of the Q-element— we observe that $B(L_a,R_a) \equiv (\neg ri, ro)\,\underline{Q}(li, lo)$, which gives the implementation of Figure 8.

16.2 *The (L/A;R)-element*

In order to generalize the preceding construction to the case of an arbitrary number of actions, we must implement the generalization of the (L/R)-

Figure 7. The D-element.



element. Sequence

$$*[S_1; S_2; \ldots; S_n] \tag{24}$$

can be decomposed into a number of shorter sequences by repeatedly applying process decomposition. There are as many ways to decompose (24) as there are binary trees of n leaves. But observe that, if $n > 2$, all decompositions will require at least one process of the form

$$(L/A; R),$$

where A and R are active communication actions. (The semicolon binds more tightly than the process call.) We shall use two different reshufflings to implement this process. Again, these reshufflings maintain the semantics of the original program if the handshaking expansion of L' is not reshuffled. The first reshuffling is

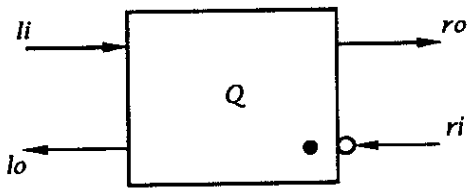
$$*[[li]; ao\uparrow; [ai]; lo\uparrow; [\neg li]; ao\downarrow; [\neg ai]; R; lo\downarrow].$$

We decompose it into two sequences by applying a process-factorization decomposition described in [10]:

$$(*[[li]; ao\uparrow; [\neg li]; ao\downarrow]$$
$$\| *[[ai]; lo\uparrow; [\neg ai]; R; lo\downarrow]$$
$$).$$

The first sequence is the wire ($li \underline{w} ao$). The second sequence is the D-element $(ai, lo) \underline{D} (ri, ro)$.

Figure 8. Implementation of the active-active buffer with a Q-element.



The second reshuffling is

$$*[[li]; A; ro\uparrow; [ri]; lo\uparrow; [\neg li]; ro\downarrow; [\neg ri]; lo\downarrow] .$$

Again, we decompose it into two sequences by process factorization:

$$(*[[ri]; lo\uparrow; [\neg ri]; lo\downarrow]$$
$$\| *[[li]; A; ro\uparrow; [\neg li]; ro\downarrow]$$
$$) .$$

The first sequence is the wire $(ri \underline{w} lo)$. The second sequence is the Q-element $(li, ro)\ \underline{Q}\ (ai, ao)$. Both implementations are shown in Figure 9.

Now the implementation of a sequence of n actions is straightforward. For instance, for $n = 4$, we have two "linear" decompositions of $(L/S_1; S_2; S_3; S_4)$. The first one is

$$((L/S_1; L_1) \| (L_1/S_2; L_2) \| (L_2/S_3; S_4)) .$$

The second one is

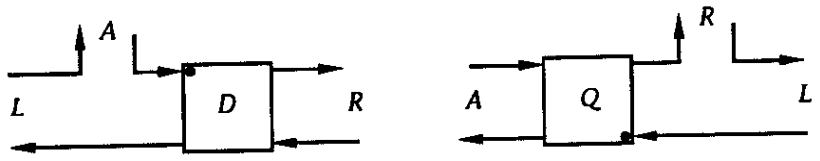
$$((L/L_2; S_4) \| (L_2/L_1; S_3) \| (L_1/S_1; S_2)) .$$

These two decompositions lead to the linear implementations shown in Figure 10.

16.3 *The Passive-Active Buffer*

In order to compose one-place buffers in a linear chain, one channel must be active and the other one passive. We implement the buffer with L passive and R active. This version is denoted by $B(L_p, R_a)$. In order to take advantage of

Figure 9. Implementations of the $(L/A; R)$-element.



the active-active case, we decompose the buffer into two processes q and t:

$$q \equiv *[D';R]$$
$$t \equiv (D/L).$$

Process q is an active-active buffer. The compilation of t is straightforward. The handshaking expansion gives

$$*[[di]; [li]; lo\uparrow; [\neg li]; lo\downarrow; do\uparrow; [\neg di]; do\downarrow].$$

Since D is an internal channel, we can reshuffle the sequence $[\neg li]; lo\downarrow$ with respect to D without introducing deadlock. (Also observe that since $do\downarrow$ remains the last action of the sequence, we have not changed the order of L relative to R.) We get

$$*[[di]; [li]; lo\uparrow; do\uparrow; [\neg di]; [\neg li]; lo\downarrow; do\downarrow].$$

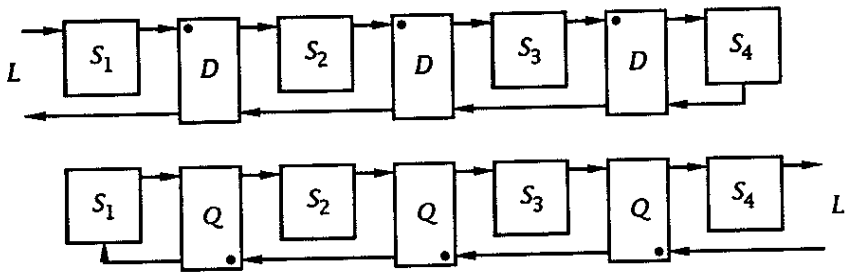
The PR expansion leading to the circuit of Figure 6 is

$$di \wedge li \mapsto lo\uparrow, do\uparrow$$
$$\neg di \wedge \neg li \mapsto lo\downarrow, do\downarrow .$$

Process t is used to connect the two ports of a channel when they are both active. It is called a "passive-passive adaptor". The complete circuit is shown in Figure 11.

The passive-active buffer can be compiled directly by introducing a state variable. The circuit obtained is slightly different. See [8].

Figure 10. Implementations of $(L/S_1;S_2;S_3;S_4)$.



17 *Single-Variable Register*

Consider the following *register* process, which provides read and write access to a simple boolean variable, x:

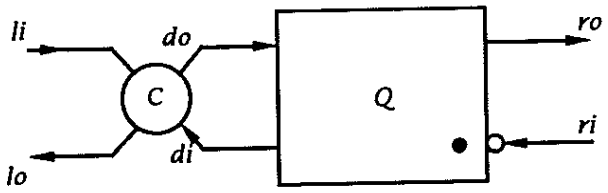
$$
\begin{array}{l}
*[[\overline{P} \rightarrow P?x \\
\quad [] \ \overline{Q} \rightarrow Q!x \\
]],
\end{array}
\tag{25}
$$

where $\neg\overline{P} \vee \neg\overline{Q}$ holds at any time.

The handshaking expansion of (25) uses the *double-rail* technique: The boolean value of x is encoded on two wires, one for the value **true** and one for the value **false**. Input channel P has two input wires, $pi1$ for receiving the value **true** and $pi2$ for receiving the value **false**, and one output wire, po. Output channel Q has two output wires, $qo1$ for sending the value **true** and $qo2$ for sending the value **false**, and one input wire, qi. Each guarded command of (25) is expanded to two guarded commands:

$$
\begin{array}{l}
*[[pi1 \rightarrow x\uparrow;\ [x];\ po\uparrow;\ [\neg pi1];\ po\downarrow \\
\quad [] \ pi2 \rightarrow x\downarrow;\ [\neg x];\ po\uparrow;\ [\neg pi2];\ po\downarrow \\
\quad [] \ x \wedge qi \rightarrow qo1\uparrow;\ [\neg qi];\ qo1\downarrow \\
\quad [] \ \neg x \wedge qi \rightarrow qo2\uparrow;\ [\neg qi];\ qo2\downarrow \\
]].
\end{array}
\tag{26}
$$

Figure 11. An implementation of the passive-active buffer.



17.1 *Mutual Exclusion between Guarded Commands*

We are now faced with a new problem: enforcing mutual exclusion between the production-rule sets of different guarded commands. (This problem is not concerned with making the *guards* of the different commands mutually exclusive. For the time being, we are considering only examples where the guards of the commands are already mutually exclusive.) Let us illustrate our problem with the compilation of the first two guarded commands. If we just concatenate the production-rule sets of these two commands, we get

$$\begin{aligned} pi1 &\mapsto x\uparrow \\ pi1 \wedge x &\mapsto po\uparrow \\ \neg pi1 &\mapsto po\downarrow \\ pi2 &\mapsto x\downarrow \\ pi2 \wedge \neg x &\mapsto po\uparrow \\ \neg pi2 &\mapsto po\downarrow . \end{aligned}$$

We now observe, however, that the second and the sixth guarded commands are interfering (they set and reset the same variable po), and that, for reasons of symmetry, the same holds for the third and the fifth PRs.

Hence, the problem of ensuring mutual exclusion between PRs of different guarded commands is the same as enforcing program order between PRs of the same guarded command. We use the same technique, which consists in strengthening the guards of the production rules, if necessary, by introducing state variables to distinguish between the states corresponding to each true guard.

In the case at hand, we strengthen the guards of the third and the sixth rules as

$$\begin{aligned} x \wedge \neg pi1 &\mapsto po\downarrow \\ \neg x \wedge \neg pi2 &\mapsto po\downarrow . \end{aligned}$$

The rest of the implementation is straightforward. The first and fourth PRs correspond to the flip-flop $(pi1, \neg pi2)\ \underline{ff}\ x$. The other PRs can be transformed into

$$\begin{aligned} (pi1 \wedge x) \vee (pi2 \wedge \neg x) &\mapsto po\uparrow \\ (\neg pi1 \wedge \neg x) \vee (\neg pi2 \wedge x) &\mapsto po\downarrow , \end{aligned}$$

which is the definition of the *if*-operator $(pi1, pi2, x)\ \underline{if}\ po$.

The production-rule expansion of the last two guarded commands of (26) gives

$$x \wedge qi \mapsto qo1\uparrow$$
$$\neg x \vee \neg qi \mapsto qo1\downarrow$$
$$\neg x \wedge qi \mapsto qo2\uparrow$$
$$x \vee \neg qi \mapsto qo2\downarrow,$$

which corresponds to the two operators $(x, qi) \vartriangle qo1$ and $(\neg x, qi) \vartriangle qo2$. The circuit is represented in Figure 12.

In the next example, we shall refer to the implementation of the first two guarded commands of (26) as the *register* operator:

$$(pi1, pi2)\ \underline{reg}\ (po, x).$$

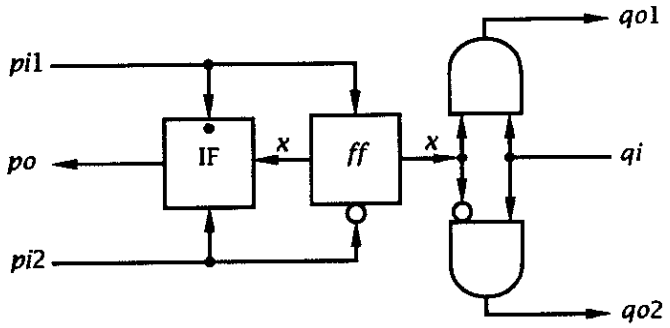
We shall refer to the implementation of the last two guarded commands of (26) as the *read* operator:

$$(qi, x)\ \underline{read}\ (qo1, qo2).$$

18 *Implementation of the Stack*

The implementation of the stack will be used to explain the general method for implementing communications that involve passing messages. The method

Figure 12. Single boolean register.



relies on the time-honored "divide-and-conquer" principle: We first construct the so-called *control part* of the program, which is the original program where the messages have been removed from each communication action. We then combine this control part with a *data path*, which is a program implementing the assignment parts of the communication actions. (See Figure 16 in Section 20.) The basic technique for combining control and data was introduced in [9].

18.1 *The Control Part of the Stack*

The control part of the stack consists of programs E and F, from which message communication has been removed. We assume that the stack is empty initially. We introduce the channel (t, t') so that F can be called from within E by process decomposition. We get

$$
\begin{aligned}
E \equiv\ & *[[\overline{in} \rightarrow in;\ t \\
& \ [] \ \overline{out} \rightarrow get;\ out \\
& \ ]]
\end{aligned}
$$

$$
\begin{aligned}
F \equiv\ & *[[\overline{t'} \wedge \overline{in} \rightarrow put;\ in \\
& \ [] \ \overline{t'} \wedge \overline{out} \rightarrow out;\ t' \\
& \ ]] .
\end{aligned}
$$

In the handshaking expansion, we let the choice of active and passive communications be dictated by the occurrence of the probes. (We will, however, return to this choice later.) We get

$$
\begin{aligned}
E \equiv\ & *[[\neg ti \wedge ini \rightarrow ino\uparrow;\ [\neg ini];\ ino\downarrow;\ to\uparrow;\ [ti];\ to\downarrow \\
& \ [] \ \neg ti \wedge outi \rightarrow geto\uparrow;\ [geti];\ geto\downarrow;\ [\neg geti];\ outo\uparrow;\ [\neg outi];\ outo\downarrow \\
& \ ]]
\end{aligned}
$$

$$
\begin{aligned}
F \equiv\ & *[[ti' \wedge ini \rightarrow puto\uparrow;\ [puti];\ puto\downarrow;\ [\neg puti];\ ino\uparrow;\ [\neg ini];\ ino\downarrow \\
& \ [] \ ti' \wedge outi \rightarrow outo\uparrow;\ [\neg outi];\ outo\downarrow;\ to'\uparrow;\ [\neg ti'];\ to'\downarrow \\
& \ ]] .
\end{aligned}
$$

Observe that, after handshaking expansion, the symmetry between E and F has been restored. The choice of whether ti or ti' should be negated in the guards determines whether E or F should be called initially, i.e., whether we start with an empty or a full stack element.

18.2 *Compilation of E*

The first guarded command, $E1$, is a standard passive-active buffer. The second guarded command, $E2$, is a standard Q-element. The implementation of E must combine the implementations of $E1$ and $E2$ in a way that enforces mutual exclusion between the execution of $E1$ and that of $E2$.

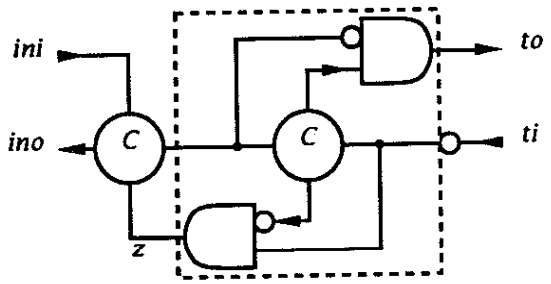
Since the execution of *in* and that of *out* are mutually exclusive, it suffices to guarantee that when *in* is completed in $E1$, $E2$ cannot start until t is completed. We introduce the variable z (initially **true**) in the handshaking expansion of $E1$, as indicated in Figure 13, and we strengthen the guard of $E2$ with z. We get

$$E1 \equiv z \wedge ini \rightarrow ino\uparrow;\ z\downarrow;\ [\neg z];\ [\neg ini];\ ino\downarrow;\ to\uparrow;\ [ti];\ to\downarrow;\ [\neg ti];\ z\uparrow,$$

$$E2 \equiv \neg ti \wedge outi \wedge z \rightarrow geto\uparrow;\ [geti];\ geto\downarrow;\ [\neg geti];\ outo\uparrow;\ [\neg outi];\ outo\downarrow .$$

Now $E2$ cannot start until $z\uparrow$ is completed, i.e., until $E1$ is completed. Since, by the structure of $E1$, $z \Rightarrow \neg ti$, we can simplify the guard of $E2$ to $outi \wedge z$. For symmetrization, we also weaken $\neg outi$ as $\neg outi \vee \neg z$. Hence, mutual exclusion is enforced by replacing input *outi* with the *and*-operator $(outi, z) \triangleq outi'$ in the Q-element implementation of $E2$. This gives the circuit of Figure 14 as an implementation of E.

Figure 13. Implementation of the first g.c. of E with variable z.



18.3 *Compilation of F*

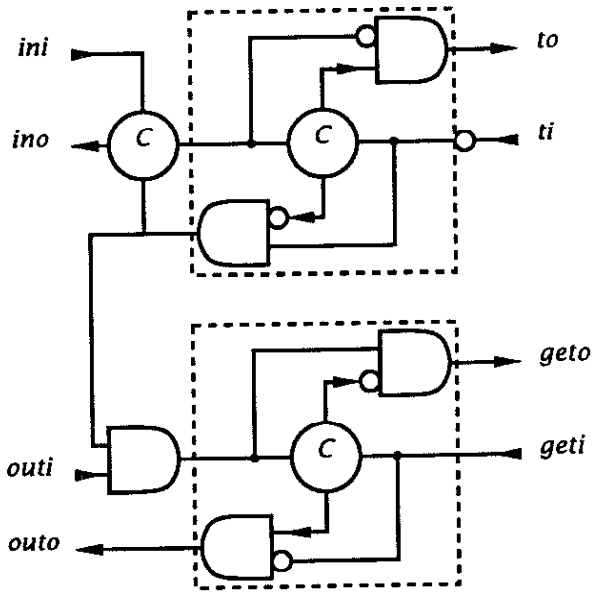
The compilation of $F1$ is identical to that of $E2$ with the appropriate change of variables. The compilation of $F2$, however, can be simplified by reshuffling. Since channel (t, t') is internal, we can reshuffle the handshaking sequence of t' without deadlock. The handshaking expansion of $F2$ becomes

$$ti' \wedge outi \rightarrow outo\uparrow;\ to'\uparrow;\ [\neg ti' \wedge \neg outi];\ outo\downarrow;\ to'\downarrow,$$

which compiles immediately into the "forked" C-element $(ti', outi)\underline{C}(outo, to')$. The reshuffling guarantees that $F1$ cannot be started before $F2$ is completed.

The channels *in* and *out* are used in both E and F, so we must merge the local copies of *in* and the local copies of *out* in a standard way that we do not describe here. The resulting circuit for the control part of the stack element is shown in Figure 15.

Figure 14. Implementation of E.

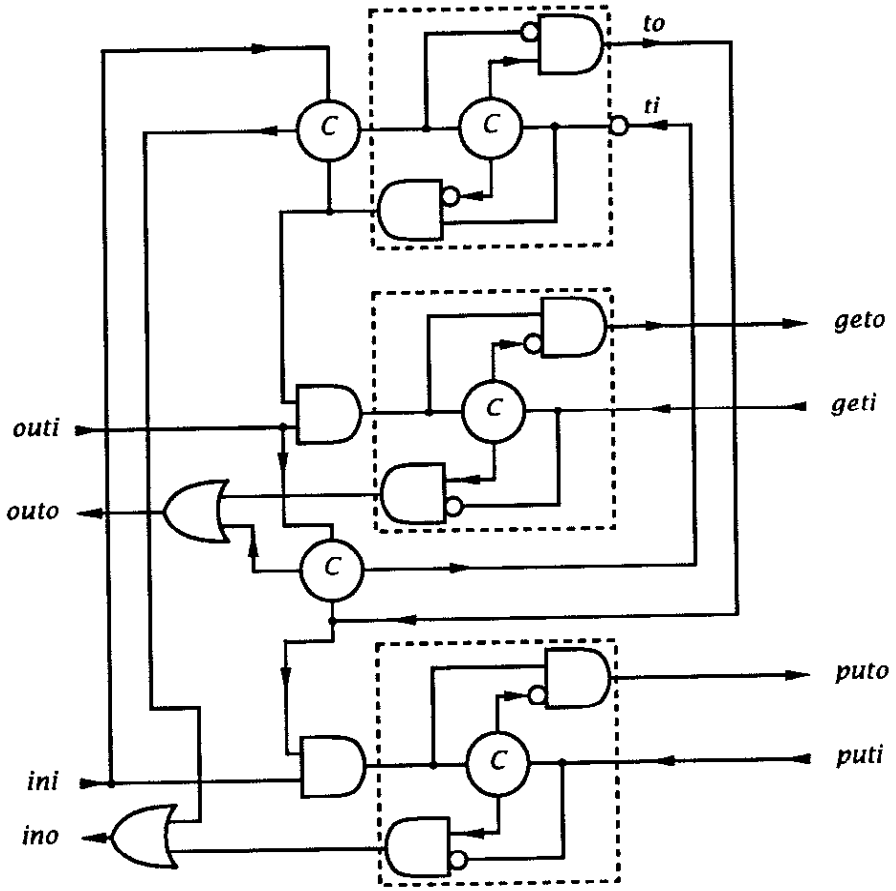


19 *Implementation of the Data Path*

We now have to extend the implementation of the control part $S2$ so as to obtain an implementation of the whole program $S1$. We want to leave $S2$ unchanged by introducing a datapath process, P, such that the parallel composition of $S2$ and P implements $S1$.

The channels *in, out, get, put* of $S2$ are renamed *in', out', get', put'*. P com-

Figure 15. The control part of the stack element.



municates with $S2$ via in', out', get', put' and with the environment via *in, out, get, put.* (See Figure 16.)

Let C be a channel of $S1$, and C' be the renamed channel of $S2$ to which C corresponds. For $(S2 \parallel P)$ to implement $S1$, each communication on C must coincide with a communication on C'; i.e., P must implement the so-called *channel interface* process

$$I_C \equiv *[C \bullet C'] .$$

Hence, P has to implement the four channel interfaces:

$* [in' \bullet in?x]$

$* [out' \bullet out!x]$

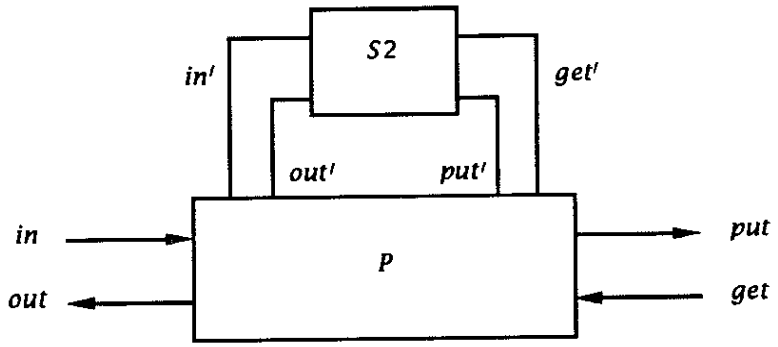
$* [get' \bullet get?x]$

$* [put' \bullet put!x] .$

20 *Implementation of Channel Interfaces*

There are four types of channel interfaces, depending on whether the port is active or passive, and whether the communication is an input or an output.

Figure 16. Adding the data path.



20.1 *Input Actions on a Passive Port*

We want to implement the interface I_C for action $C?x$ on the passive port C. I_C communicates with $S2$ by the active port C', and with the environment by the passive port D. Furthermore, in the standard double-rail encoding technique, the two-wire implementation (ci, co) of C has to be interfaced to the three-wire input port D in which the two input wires, $di1$ and $di2$, are used to encode the two values of the incoming message. (See Figure 17.)

I_C has to implement an interleaving of the following three sequences:

$$S_C \equiv *[ci'\uparrow;\ [co'];\ ci'\downarrow;\ [\neg co']]$$

$$S_D \equiv *[[di1 \vee di2];\ do\uparrow;\ [\neg di1 \wedge \neg di2];\ do\downarrow]$$

$$S_X \equiv *[[di1 \rightarrow x\uparrow;\ [x]\ []\ di2 \rightarrow x\downarrow;\ [\neg x]]]\,.$$

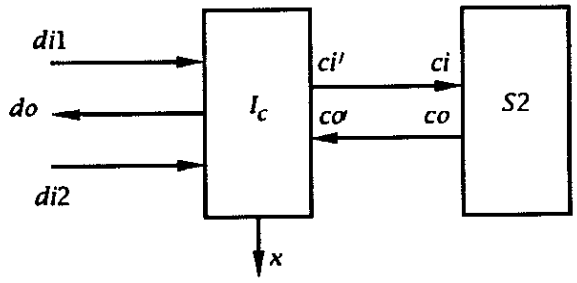
An implementation of $C' \bullet D$ interleaves sequences S_C and S_D as

$$*\,[[di1 \vee di2];\ ci'\uparrow;\ [co'];\ do\uparrow;\ [\neg di1 \wedge \neg di2];\ ci'\downarrow;\ [\neg co'];\ do\downarrow]\,. \tag{28}$$

In the interleaving of (28) and S_X, the assignment to x is inserted after $[co']$ so as to ensure that communication action C has been started when the assignment to x is performed:

$$\begin{aligned} *[[di1 \vee di2];\ ci'\uparrow;\ [co' \wedge di1 \rightarrow x\uparrow;\ [x]\,[]\ co' \wedge di2 \rightarrow x\downarrow;\ [\neg x]]; \\ do\uparrow;\ [\neg di1 \wedge \neg di2];\ ci'\downarrow;\ [\neg co'];\ do\downarrow]\,. \end{aligned} \tag{29}$$

Figure 17. Channel interface for input port.



Next, we factor (29) as

$$*\,[[di1 \vee di2];\ ci'\uparrow;\ [\neg di1 \wedge \neg di2];\ ci'\downarrow] \tag{30}$$

and

$$\begin{array}{l} *[[co' \wedge di1 \rightarrow x\uparrow;\ [x];\ do\uparrow;\ [\neg co'];\ do\downarrow \\ \quad [\!]\ co' \wedge di2 \rightarrow x\downarrow;\ [\neg x];\ do\uparrow;\ [\neg co'];\ do\downarrow \\ \quad]]\,. \end{array} \tag{31}$$

Sequence (30) is realized by the operator $(di1, di2)\ \underline{\vee}\ ci'$. We factor (31) so as to isolate the register part:

$$\begin{array}{rcl} (co', di1)\ \underline{aC}\ x1 & \equiv & *[[co' \wedge di1];\ x1\uparrow;\ [\neg co'];\ x1\downarrow] \\ (co', di2)\ \underline{aC}\ x2 & \equiv & *[[co' \wedge di2];\ x2\uparrow;\ [\neg co'];\ x2\downarrow] \\ (x1, x2)\ \underline{reg}\ (x, do) & \equiv & *[[x1 \rightarrow x\uparrow;\ [x];\ do\uparrow;\ [\neg x1];\ do\downarrow \\ & & \quad [\!]\ x2 \rightarrow x\downarrow;\ [\neg x];\ do\uparrow;\ [\neg x2];\ do\downarrow \\ & & \quad]]\,. \end{array}$$

The implementation is shown in Figure 18.

20.2 *Input Actions on an Active Port*

For port C active, the communication variables of the interface I_C remain the same. But now the handshaking expansions of C' and D are different, since C' is passive and D is active. We get

$$\begin{array}{rcl} S_C & \equiv & *[[co'];\ ci'\uparrow;\ [\neg co'];\ ci'\downarrow] \\ S_D & \equiv & *[do\uparrow;\ [di1 \vee di2];\ do\downarrow;\ [\neg di1 \wedge \neg di2]] \\ S_X & \equiv & *[[di1 \rightarrow x\uparrow;\ [x]\ [\!]\ di2 \rightarrow x\downarrow;\ [\neg x]]]\,. \end{array}$$

(Observe that S_X is not changed.) An interleaving of S_C and S_D that implements $C' \bullet D$ is the interleaving corresponding to two wires:

$$*[[co'];\ do\uparrow;\ [di1 \vee di2];\ ci'\uparrow;\ [\neg co'];\ do\downarrow;\ [\neg di1 \wedge \neg di2];\ ci'\downarrow]\,.$$

As to the implementation of the assignment to x, we now observe that, since C and D are active, there is no risk of the assignment to x being started before

C is. The interleaving obtained is

$$*[[co']; do\uparrow; \quad [di1 \rightarrow x\uparrow \;[]\; di2 \rightarrow x\uparrow]; \quad ci'\uparrow; [\neg co']; do\downarrow; [\neg di1 \wedge \neg di2]; ci'\downarrow], \tag{32}$$

which can be factored into the wire

$$(co' \underline{w}\, do) \equiv *[[co']; do\uparrow; [\neg co']; do\downarrow]$$

and the register

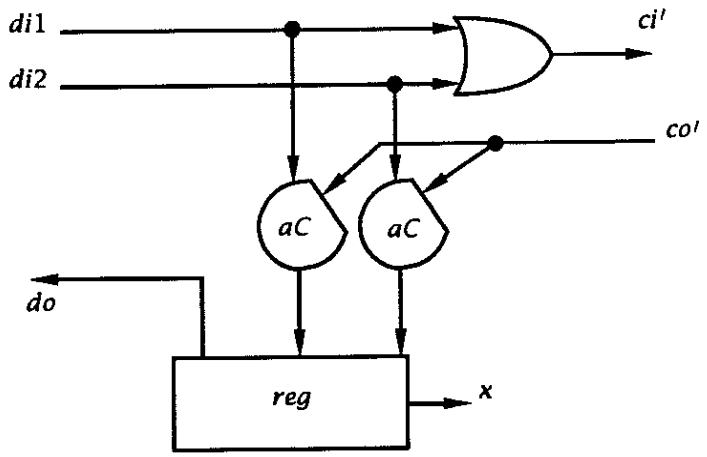
$$(di1, di2)\, \underline{reg}\, (x, ci') \equiv *[[di1 \rightarrow x\uparrow; [x]; ci'\uparrow; [\neg di1]; ci'\downarrow \;[]\; di2 \rightarrow x\downarrow; [\neg x]; ci'\uparrow; [\neg di2]; ci'\downarrow \;]].$$

The implementation of the interface is shown in Figure 19.

20.3 *Output Actions*

In the case of an output, like $out!x$ or $put!x$, the implementation turns out to be the same for passive and active ports. Given the same nomenclature as in

Figure 18. Input actions on passive port.



the input case, port D is now implemented with two output variables, $do1$ and $do2$, and one input variable, di. Port C' is not changed. The rest of the derivation is straightforward and is left as an exercise for the reader. It leads to a wire and a *read* operator, which we have introduced in the implementation of the register:

$$di \underline{w} cin \equiv *[[di];\ ci'\uparrow;\ [\neg di];\ ci'\downarrow]$$

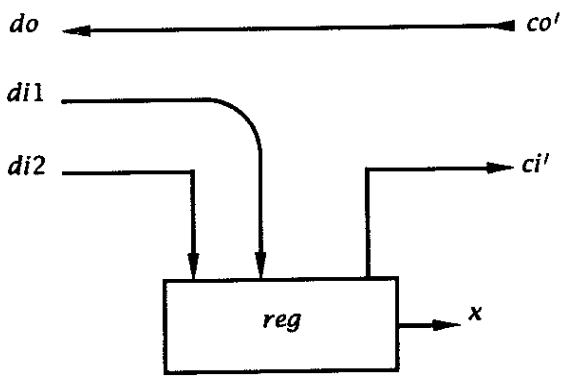
$$(co', x) \underline{read} (do1, do2) \equiv *[[x \wedge co' \rightarrow do1\uparrow;\ [\neg co'];\ do1\downarrow$$
$$[]\ \neg x \wedge co' \rightarrow do2\uparrow;\ [\neg co'];\ do2\downarrow$$
$$]].$$

The only difference between the active and the passive cases is that, in the active case, the *read* is activated first. In the passive case, the wire is activated first. The circuit is shown in Figure 20.

20.4 *Active Input and Passive Output*

A somewhat surprising result of this implementation of input and output commands is that, contrary to common belief, it is simpler to implement input commands with active ports than with passive ports. The gain is quite

Figure 19. Input actions on active port.



important: For n bits of data, the active implementation saves $2 \times n$ asymmetric C-elements and n *or*-gates. On the other hand, the implementation of output actions is the same for active and passive ports.

Therefore, we shall always implement input actions with active ports. When the input port is probed, like *in* in the stack example, we shall use a slightly more complicated implementation of the handshaking protocol that makes it possible to probe an active port.

20.5 *Lazy-Active Protocol*

Consider the active implementation of communication command X:

$$xo\uparrow; [xi]; xo\downarrow; [\neg xi] .$$

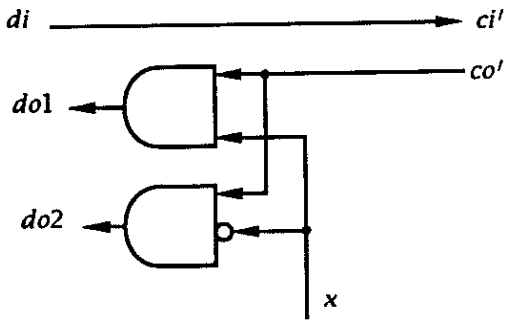
We introduce an alternative active protocol, called *lazy-active*:

$$[\neg xi]; xo\uparrow; [xi]; xo\downarrow .$$

The lazy-active protocol is derived from the active one by postponing wait action $[\neg xi]$ until the beginning of the next communication on X, and by adding a vacuous wait action $[\neg xi]$ at the beginning of the first communication X. Hence, the lazy-active protocol is a correct implementation.

Consider sequence $X;S$, where S is an arbitrary program part. With X lazy-active, half of the communication delays overlap with the execution of S. The

Figure 20. Output-action interface.



gain is particularly important when data communication is involved, since half of the data-transmission delays and half of the "completion-tree" delays can overlap with the rest of the computation.

This important property of lazy-active protocols was discovered recently by Steve Burns. All input actions are now implemented as lazy-active. We have not done so in the stack, which is an older design.

21 *The Complete Circuit for the Stack*

The sharing of register *x* by ports *in* and *get* has to be implemented either by a multiplexer or by a multiport flip-flop. Since only two ports share the register, we choose to use a dual-port flip-flop. The complete datapath is shown in Figure 21.

The complete circuit obtained by composing the different parts together is shown in Figure 22. An important optimization has been added to the

Figure 21. The complete datapath.

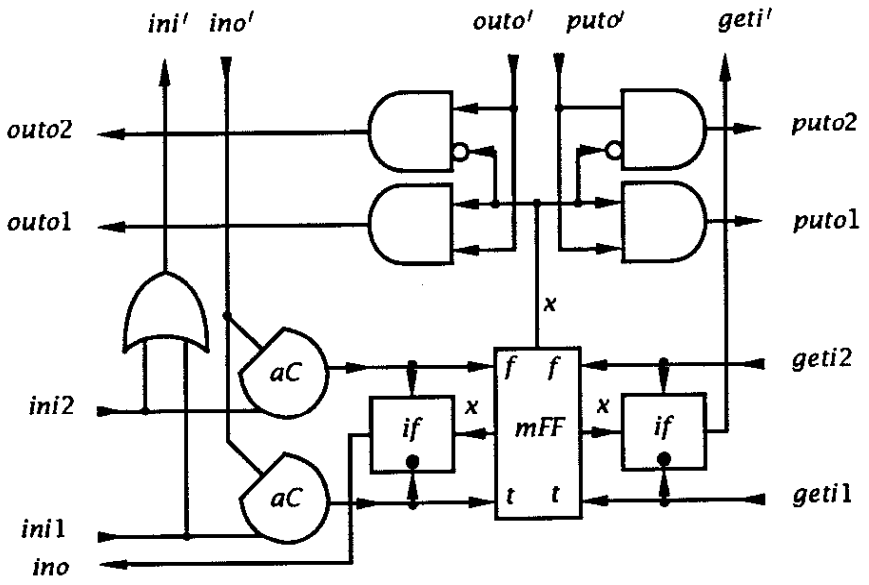
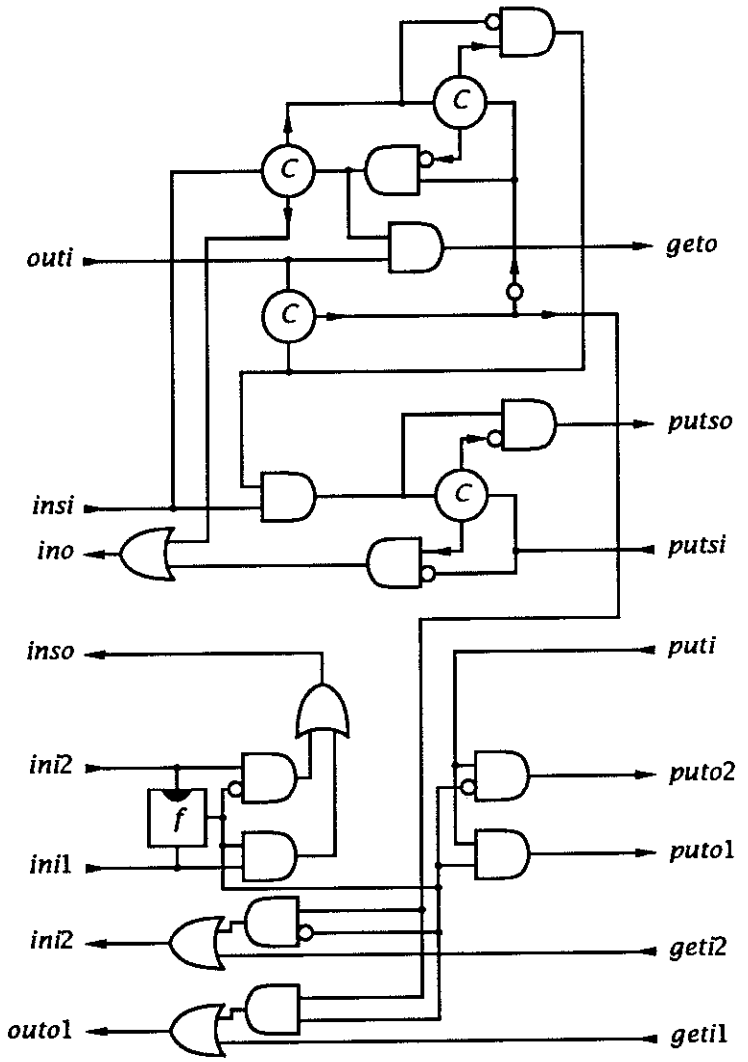


Figure 22. The complete circuit for a one-bit stack element.



design. It concerns the implementation of the second guard of E:

$$\overline{out} \rightarrow get?x;\ out!x\,.$$

We observe that the value of x involved in the second action ($out!x$) is the same as the value of x involved in the first action ($get?x$). We can therefore encode the transmitted value in the handshaking expansion of the guarded command without having to use register x. We are tempted to make this optimization available to the programmer by allowing assignments to ports. We would then write

$$\overline{out} \rightarrow out!get\,.$$

The preceding modification leads to a significant simplification of the circuit since we can eliminate a D-element, and, *for each bit of the data path*, we can eliminate an IF-element and replace the multiport flip-flop with a simple flip-flop. The chip we have fabricated includes this modification, as well as the optimization that consists in making input port *in* active.

22 *A Delay-Insensitive Fair Arbiter*

This last example addresses the issues of arbitration between guards and unstable guards. We have already discussed the metastability property of arbiters. The realization of a delay-insensitive arbiter, however, raises another issue: fairness. An arbiter is *strongly fair* when a pending communication request is granted after a bounded number of other requests are granted. An arbiter is *weakly fair* when a request is granted after a finite but possibly unbounded number of other requests. Whether it is possible to construct a delay-insensitive fair arbiter has been, so far, an open question. It has been conjectured that delay-insensitive fair arbiters do not exist. In this example, we prove the existence of delay-insensitive fair arbiters by constructing one.

22.1 *A Fair-Arbiter Program*

The process *fsel* described in the first part defines a fair arbitration program between two unrelated inputs. We choose to implement the following simplified version of *fsel*:

$$*\,[[\overline{A} \rightarrow A[]\ \neg\overline{A} \rightarrow \textbf{skip}];[\overline{B} \rightarrow B[]\ \neg\overline{B} \rightarrow \textbf{skip}]]\,. \tag{33}$$

According to (33), when $\overline{A}$ holds, A will be completed after at most one B action, regardless of the current state of the computation. Hence, the arbiter is strongly fair towards requests A and B. Assume that A' is pending at a certain

point of the computation. By definition of the probe, $\overline{A}$ is **true** eventually; i.e., a finite but unbounded number of B actions can be completed between the moment $\mathbf{q}A'$ holds and the moment $\overline{A}$ holds. Hence, the arbiter is only *weakly* fair towards requests A' and B'.

Therefore, *with this definition of suspension of an action*, we can say that the arbiter is strongly fair towards requests that have reached the arbiter and weakly fair towards all requests. (We could redefine the suspension of a communication action X such that $\mathbf{q}X$ holds only when the initiation of action X can be observed by the other process. With this definition of suspension, we have $\mathbf{q}A' = \overline{A}$. The arbiter is then strongly fair towards all requests.)

22.2 *The Compilation*

Applying the process decomposition rule, we decompose (33) into three processes ($P1 \parallel P2 \parallel P3$). Channels (C, D) between $P1$ and $P2$, and (E, F) between $P1$ and $P3$ are introduced:

$$
\begin{aligned}
P1 &\equiv *[E; C] \\
P2 &\equiv *[[\overline{D} \wedge \overline{B} \rightarrow B; D \\
&\quad [] \overline{D} \wedge \neg\overline{B} \rightarrow D \\
&\quad]] \\
P3 &\equiv *[[\overline{F} \wedge \overline{A} \rightarrow A; F \\
&\quad [] \overline{F} \wedge \neg\overline{A} \rightarrow F \\
&\quad]] .
\end{aligned}
$$

Ports D and F are implemented as passive; ports C and E are implemented as active. Hence $P1$ is the standard active-active buffer. The handshaking expansion of $P2$ gives

$$
\begin{aligned}
P2 \equiv\ & *[[di \wedge bi \rightarrow bo\uparrow;\ [\neg bi];\ bo\downarrow;\ do\uparrow;\ [\neg di];\ do\downarrow \\
& \quad [] di \wedge \neg bi \rightarrow do\uparrow;\ [\neg di];\ do\downarrow \\
& \quad]] .
\end{aligned}
$$

Because bi can change from **false** to **true** asynchronously, the second guard of $P2$ is not stable; i.e., its value can change from **true** to **false** at any time.

In order to make both guards of $P2$ stable, we introduce the synchronizer

$$\begin{aligned} sync \equiv\ & *[[di \wedge bi \rightarrow u\uparrow;\ [\neg di];\ u\downarrow \\ & \ [] \ di \wedge \neg bi \rightarrow v\uparrow;\ [\neg di];\ v\downarrow \\ & \ ]]\,. \end{aligned}$$

Sync is the standard operator we have described in Part I. We now have to find a process, X, such that $(X \| sync) = P2$. Since *sync* is entirely defined, we would like to be able to perform the inverse operation of $\|$, or "process quotient", so as to compute X as $X = (P2 \div sync)$. A way to perform this quotient is to remove all actions of *sync* from $P2$, and then to check whether the result fulfills $(X \| sync) = P2$.

To perform the quotient as suggested, $P2$ should be extended to contain all actions of *sync*, so that the orders of actions are compatible in *sync* and in the extended version of $P2$. (This procedure is explained in [10].) The extension of $P2$ gives

$$\begin{aligned} & *[[\ di \wedge bi \rightarrow u\uparrow;\ [u];\ bo\uparrow;\ [\neg bi];\ bo\downarrow;\ do\uparrow;\ [\neg di];\ u\downarrow;\ [\neg u];\ do\downarrow \\ & \ [] \ di \wedge \neg bi \rightarrow v\uparrow;\ [v];\ do\uparrow;\ [\neg di];\ v\downarrow;\ [\neg v];\ do\downarrow \\ & \ ]]\,. \end{aligned}$$

We obtain for X

$$\begin{aligned} & *[[u \rightarrow bo\uparrow;\ [\neg bi];\ bo\downarrow;\ do\uparrow;\ [\neg u];\ do\downarrow \\ & \ [] \ v \rightarrow do\uparrow;\ [\neg v];\ do\downarrow \\ & \ ]]\,. \end{aligned}$$

The compilation of the first guarded command is facilitated if transition $bo\downarrow$ is postponed until after $[\neg u]$. This transformation does not introduce deadlock since the completion of D does not depend on the completion of B. After this transformation, the PR expansion gives

$$\begin{array}{rclrcl} u & \mapsto & bo\uparrow & \neg u & \mapsto & bo\downarrow \\ u \wedge \neg bi & \mapsto & do\uparrow & v & \mapsto & do\uparrow \\ bi \vee \neg u & \mapsto & do\downarrow & \neg v & \mapsto & do\downarrow\,. \end{array}$$

The operator reduction, which includes the introduction of auxiliary variables

do' and do'', gives

$$\begin{aligned} u \ &\underline{w}\ bo \\ (u, \neg bi) \ &\underline{\wedge}\ do' \\ v \ &\underline{w}\ do'' \\ (do', do'') \ &\underline{\vee}\ do\,. \end{aligned}$$

The circuit is shown in Figure 23. The implementation of $P3$ is identical.

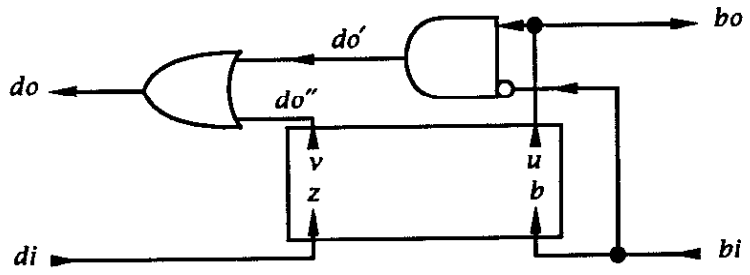
22.3 *The Circuit*

The final circuit, shown in Figure 24, is obtained by composing the two identical circuits implementing $P2$ and $P3$ with the circuit of $P1$. The reshuffled version of $P1$, consisting of a wire and an inverter, can also be used if it can be proved that the reshuffling does not introduce deadlock. The circuit shown in Figure 24 includes a minor optimization that eliminates the negated inputs that are also the output of a fork.

23 *Conclusion*

We have described a method for implementing a concurrent program (a set of communicating processes) as a network of digital operators that can be directly mapped into a delay-insensitive VLSI circuit. The circuit is derived from the program by applying a series of systematic, semantics-preserving

Figure 23. Implementation of $P2$.



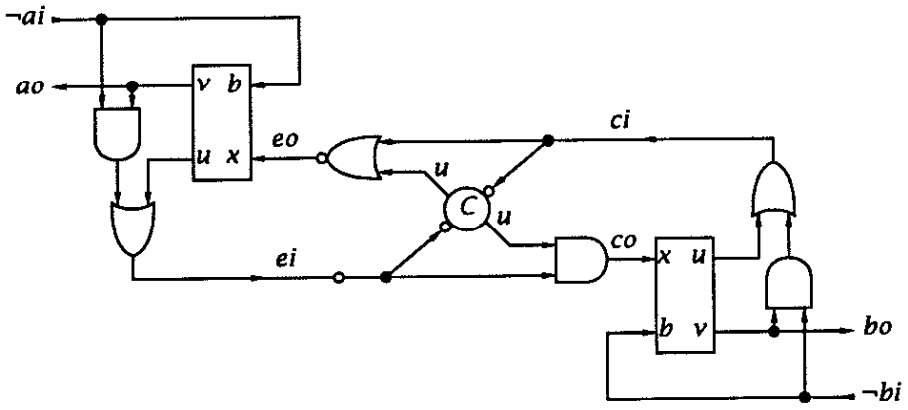
transformations that we have compared to compiling. Hence, the circuits are correct by construction, and their logical correctness is independent of the delays in operators and wires, with the exception of isochronic forks.

The examples cover most of the constructs of the language but not all of them: We have not shown how to implement an arbitrary set of guards. Therefore, we have not quite shown that *any* program in the language can be compiled. Such a proof has been given in [1] and [2], where the compilation of each construct is described as part of the basic algorithm for an automatic compiler. It is shown that any program in a subset of the language can be implemented as a delay-insensitive circuit using only a small set of basic elements: the two-input C-element, the two-input or-gate or two-input and-gate, the synchronizer, the inverter, and the isochronic fork.

There is no reason, however, for confining the designer to a minimal set of operators. On the contrary, since an advantage of VLSI is the possibility to create operators at no cost, introducing the special-purpose operator that exactly implements an arbitrary set of production rules often simplifies a circuit drastically.

In order to convince the VLSI community of the practicality of our method, it was essential to fabricate the circuits we had designed. Hence, all significant

Figure 24. Implementation of the fair arbiter.



examples that we have used in our research —distributed mutual exclusion, queues, stacks, routing automata for a communication network, the $3X + 1$ engine— have been fabricated in SCMOS using the MOSIS foundry service. They have all be found to be correct on "first silicon". They are also very robust and —given the low level of circuit optimization applied— surprisingly fast. The $3x + 1$ engine, constructed by Tony Lee, is a special-purpose processor consisting of a state-machine and an 80-bit-wide datapath. It contains approximately 40,000 transistors and operates at over 8 MIPS (million instructions per second) in 2μm MOSIS SCMOS technology.

At the moment of writing, we have just completed the design of the first asynchronous general-purpose microprocessor [12]. It is a 16-bit RISC-like architecture with independent instruction and data memories. It has 16 registers, four buses, an ALU, and two adders. The size is about 20,000 transistors. Two versions have been fabricated: one in 2μm MOSIS SCMOS, and one in 1.6μm MOSIS SCMOS. (On the 2μm version, only 12 registers were implemented in order to fit the chip on an 84-pin 6600μm $\times$ 4600μm package.)

The chips are entirely delay-insensitive, with the sole exception of the interface with the memories and, of course, the isochronic forks. In the absence of available memories with asynchronous interfaces, we have simulated the completion signal from the memories with an external —off-chip— delay. For testing purposes, the delay on the instruction memory interface is variable.

In spite of the presence of floating *n*-wells, the 2μm version runs at 12 MIPS. The 1.6μm version runs at 18 MIPS. (Those performance figures are based on measurements from sequences of ALU instructions without carry. They take no advantage of the overlap between ALU and memory instructions.) Those performances are quite encouraging given that the design is very conservative: no pass-transistors, static gates, dual-rail encoding of data, completion trees, etc.

Only 2 of the 12 2μm chips passed all tests, but 34 of the 50 1.6μm chips were found to be entirely functional.

We have tested the chips under a wide range of *VDD* voltage values. At room temperature, the 2μm version is functional in a voltage range from 7V down to 1V! It reaches 15 MIPS at 7V. We have also tested the chips cooled in liquid nitrogen. The 2μm version reaches 20 MIPS at 5V and 30 MIPS at 12V. The 1.6μm version reaches 30 MIPS at 5V. Of course, these measurements are made without adjusting any clocks (there are none), but simply by connecting the processor to a memory containing a test program and observing the rate of instruction execution. The power consumption is 145mW at 5V, and 6.7mW at 2V.

24 *Acknowledgments*

I am indebted to my students Steve Burns, Dražen Borković, Pieter Hazewindus, Tony Lee, Marcel van der Goot, José Tierno, and Kevin Van Horn for their contributions to the research and their comments on the manuscript. Acknowledgments are also due to Chuck Seitz, Jan van de Snepscheut, Martin Rem, and Huub Schols for numerous discussions on the topic.

References

[1] Burns, S. M. "Automated compilation of concurrent programs into self-timed circuits". Technical Report CS-TR-88-2, M.S. Thesis, Computer Science Department, California Institute of Technology, 1988.

[2] Burns, S. M. and Martin, A. J. "Syntax-directed translation of concurrent programs into self-timed circuits". *Proceedings of the Fifth MIT Conference on Advanced Research in VLSI*, J. Allen and F. Leighton, eds., pp. 35–40. MIT Press, Cambridge, Mass., 1988.

[3] Dijkstra, Edsger W. *A Discipline of Programming.* Prentice-Hall, Englewood Cliffs, N.J., 1976.

[4] Hoare, C. A. R. "Communicating sequential processes". *Communications of the ACM 21*, 8 (August 1978), pp. 666–677.

[5] Martin, A. J. "The probe: An addition to communication primitives". *Information Processing Letters 20* (1985), pp. 125–130.

[6] Martin, A. J. "Compiling communicating process into delay-insensitive VLSI circuits". *Distributed Computing 1*, 4 (1986).

[7] Martin, A. J. "A delay-insensitive fair arbiter". Technical Report 5193:TR:85, Computer Science Department, California Institute of Technology, 1985.

[8] Martin, A. J. "FIFO: An exercise in compiling programs into circuits". In *From HDL Description to Guaranteed Correct Circuit Design*, D. Borrione, ed. North-Holland, Amsterdam, 1986.

[9] Martin A. J. "A synthesis method for self-timed VLSI circuits". *ICCD 87: 1987 IEEE International Conference on Computer Design*, pp. 224–229. IEEE Computer Society Press, Los Alamitos, Calif., 1987.

[10] Martin, A. J. "Formal program transformations for VLSI circuit synthesis". In *Formal Development of Programs and Proofs*, E. W. Dijkstra, ed. Addison-Wesley, Reading, Mass., 1989.

[11] Martin, A. J. "The limitations to delay-insensitivity in asynchronous circuits". *Proceedings of the Sixth MIT Conference on Advanced Research in VLSI*, W. J. Dally, ed. MIT Press, Cambridge, Mass., 1990.

[12] Martin, A. J., Burns, S. M., Lee, T.K., Borkovic, D., and Hazewindus, P. J. "The design

of an asynchronous microprocessor". *Decennial Caltech Conference on VLSI*, C. L. Seitz, ed., pp. 351–373. MIT Press, Cambridge, Mass., 1989.

[13] May, D. "Compiling occam into silicon". This volume (Chapter 3).

[14] Mead, C. and Conway, L. *Introduction to VLSI Systems.* Addison-Wesley, Reading, Mass., 1980.

[15] Miller, R. E. *Switching Theory*, Vol. 2. Wiley, New York, 1965.

[16] Seitz, C. L. "System timing." *Introduction to VLSI systems.* Chapter 7 of [14].

[17] Snepscheut, J. v. d. *Trace Theory and VLSI Design.* Lecture Notes in Computer Science, vol. 200. Springer-Verlag, Berlin, 1985.

[18] Weste, N. and Eshraghian, K. *Principles of CMOS VLSI Design.* Addison-Wesley, Reading, Mass., 1985.